Universidad
Carlos III de Madrid

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

OPTIMIZACIÓN Y EVALUACIÓN DEL BENCHMARK PARSEC MEDIANTE INTEL ARRAY BUILDING BLOCKS

Autor: Sergio Rodríguez Martos

Tutor: José Daniel García Sánchez

Colmenarejo, Septiembre de 2012

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mi tutor José Daniel García Sánchez por su paciencia y el esfuerzo invertido. También quiero agradecer al resto de profesores por su inestimable ayuda durante la carrera que me han permitido llegar hasta este punto.

A mis padres y mi hermana, porque han sabido esperar y gracias a ellos he podido llegar a ser quien soy ahora. Ellos me han dado todo lo que ha estado a su alcance.

A mi novia, por saber apoyarme y aguantarme cuando las cosas se ponían cuesta arriba y no les veía solución.

A mis amigos de siempre. Gracias a ellos he podido dejar el proyecto a un lado cuando necesitaba tomarme un respiro y divertirme sin importar el cuándo, el cómo y el qué.

Por último, agradecer a todos mis compañeros y amigos de la universidad que siempre me han ayudado en el día a día. Las largas sesiones de biblioteca ahora se verán recompensadas.

Gracias a todas estas personas, y alguna más que me dejo en el tintero, he conseguido dar un pasito más en mi carrera personal y avanzar hacia mis objetivos.

RESUMEN

Este Trabajo Fin de Grado pretende presentar Intel Array Building Blocks como una nueva alternativa a los modelos de programación paralela existentes en arquitecturas de memoria compartida. Para ello, se han evaluado las trece aplicaciones contenidas en el benchmark PARSEC, analizando cuales de ellas podrían ofrecer un buen rendimiento. Con el fin de aplicar esta nueva tecnología se han seleccionados dos de los programas del benchmark, Blackscholes y Fluidanimate. El desarrollo se ha realizado en C++ a partir de los códigos secuenciales de las aplicaciones.

Intel Array Building Blocks es una biblioteca basada en el lenguaje C++ que proporciona paralelismo de datos mediante la combinación de varios núcleos e instrucciones vectoriales en arquitecturas multicore. ArBB está orientado a la optimización de operaciones matriciales y vectoriales.

El benchmark PARSEC ha sido concebido con fines tanto académicos como científicos y ofrece un conjunto de aplicaciones para arquitecturas de memoria compartida. Estas aplicaciones, de ámbito muy diverso, han sido paralelizadas previamente con tecnologías conocidas como Pthreads, OpenMP o Intel TBB.

Blackscholes se basa en la resolución de una conocida ecuación del ámbito financiero mediante la realización de cálculos intensivos de operaciones matemáticas. La optimización de esta aplicación se ha realizado utilizando íntegramente Intel Array Building Blocks.

Fluidanimate se encarga de simular la dinámica del movimiento de un fluido. Esta aplicación de animación ha sido paralelizada con Intel ArBB junto con Intel TBB debido a las características específicas del algoritmo y las restricciones impuestas en Array Building Blocks.

La evaluación del rendimiento de las dos aplicaciones paralelizadas se ha realizado sobre un computador con 8 hilos de ejecución y memoria uniforme (UMA) y un computador con 48 hilos de ejecución y memoria no uniforme (NUMA). Una vez paralelizada la aplicación, los resultados obtenidos de Blackscholes son mejores a los del resto de modelos de programación paralela (Pthreads, OpenMP e Intel TBB), logrando una aceleración máxima de 13,40 para la arquitectura con 8 hilos de ejecución y de 22,11 para la arquitectura con 48 hilos de ejecución. Por su parte, los resultados obtenidos en la versión implementada con ArBB y TBB de Fluidanimate superan, en la mayoría de pruebas realizadas, a las versiones optimizadas con TBB y Pthreads incluidas en el benchmark PARSEC. La aceleración máxima obtenida es de 3,21 para la arquitectura con 8 hilos de ejecución y de 18,45 para la arquitectura con 48 hilos de ejecución

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

ÍNDICE DE GRÁFICOS

1. INTRODUCCIÓN

Este primer punto muestra una visión general del trabajo fin de grado realizado. En él se justifica la elección del proyecto, los objetivos planteados en un inicio y la estructura que compone el documento, además de una tabla de acrónimos para facilitar la lectura.

1.1.Motivación

El aumento del rendimiento de los procesadores ha sido el objetivo principal de los ingenieros desde hace décadas. Esta progresión se ha cumplido sin problemas hasta los inicios del siglo XXI gracias al aumento del número de transistores, de la frecuencia del reloj y del ancho de palabra del procesador, además de la utilización del paralelismo a nivel de instrucción mediante técnicas como la ejecución fuera de orden o el uso de predictores de saltos. El aumento, año tras año, del número de transistores se ha cumplido siguiendo la tendencia que enunció Moore en su ley del año 1695, la cual fue rectificada más tarde en el año 1975, en la que afirmó que el número de transistores sigue un crecimiento exponencialmente duplicando el número de elementos cada 24 meses. Este aumento se ha conseguido gracias a la reducción del tamaño de las puertas lógicas pero, debido a los límites físicos de los semiconductores utilizados, han surgido problemas difíciles de superar, tales como la disipación de calor generado, el aumento del consumo energético o la necesidad de sincronizar la información. Con el fin de intentar lograr un mejor rendimiento en los procesadores secuenciales, se agregaron instrucciones vectoriales que utilizaban el modelo SIMD (*Single Instruction Multiple Data*) que se basa en realizar la misma operación sobre un conjunto de datos. Inicialmente, Intel desarrolló en 1997 el juego de instrucciones MMX de 64 bits, técnica que ha ido evolucionando logrando juegos de instrucciones de 256 bits como el conocido AVX. Esta mejora no fue suficiente y la viabilidad de esta arquitectura tuvo que ser reemplazada por una nueva generación de procesadores para así continuar con la tendencia del aumento del rendimiento.

Como se muestra en la Ilustración 1, hace alrededor de una década, los procesadores adquirieron el límite en cuanto a la frecuencia del procesador, el consumo de energía o la paralelización a nivel de instrucción. Así pues, se inició una revolución en la industria diseñando arquitecturas multicore que combinan dos o más procesadores en un único socket. Con el fin de aprovechar esta nueva generación de procesadores, la aparición de bibliotecas de funciones que extienden de un lenguaje secuencial, como POSIX Threads, o de lenguajes paralelos, como OpenMP, han sido una de las opciones más recurrentes para explotar la concurrencia en las aplicaciones paralelizadas. Para evaluar estas tecnologías, surgieron benchmarks que permiten conocer el rendimiento de una aplicación en procesadores multicore.

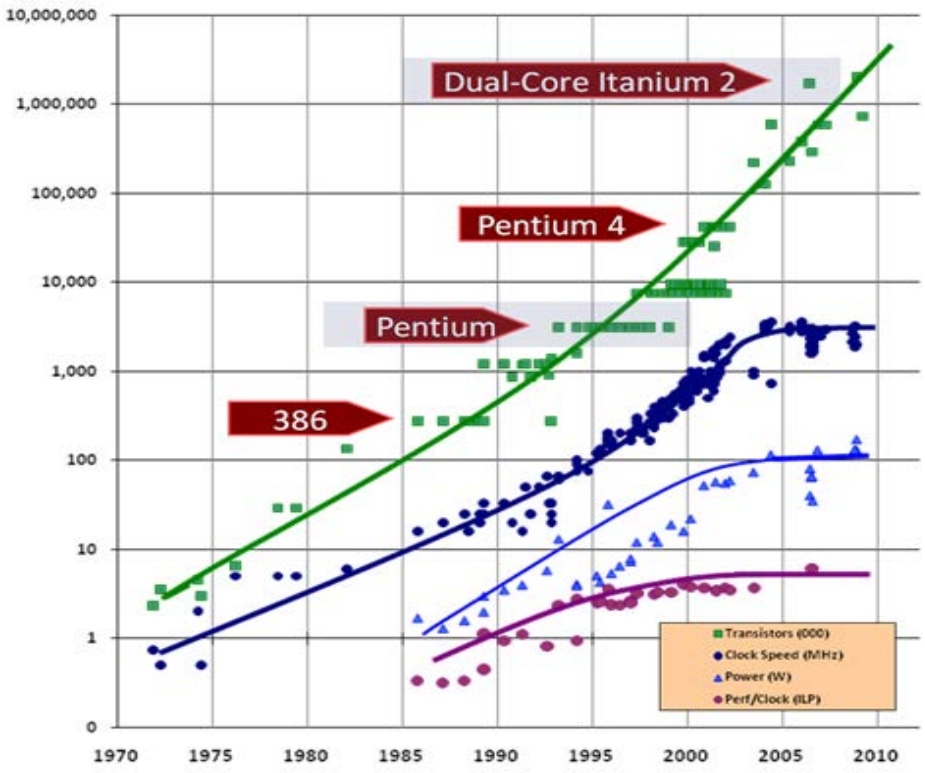


Ilustración 1. Tendencias de los procesadores [1]

La programación paralela suele ser una tarea compleja, ya que es necesario tener en cuenta que surgen determinados problemas, tales como la escalabilidad, la sincronización entre procesadores o los bloqueos generados. Como consecuencia, el programador debe preocuparse por una gran cantidad de aspectos además del propio algoritmo del programa. La laboriosa tarea de paralelizar una aplicación ha dificultado la expansión de estas tecnologías. Además, la mejora del rendimiento parece haberse estancado en estos últimos años. Fruto de los problemas que ocasiona la programación paralela para el desarrollador y la baja tasa de aumento del rendimiento, ha surgido un nuevo modelo, desarrollado por Intel, denominado Intel Array Building Blocks, que se encarga de combinar la utilización de varios cores de forma simultánea con la utilización de instrucciones vectoriales.

Array Building Blocks se aleja de las tecnologías convencionales usadas hasta la fecha, ya que permite un mayor rendimiento y una mayor sencillez en la resolución de problemas de computación paralela. Este tipo de modelo puede ser considerado dentro de la nueva generación de tecnologías para la paralelización de aplicaciones en aras de obtener una mejor tasa de rendimiento, por lo que es interesante evaluar su comportamiento con un benchmark para arquitecturas multi-core.

A todo lo descrito anteriormente, he de añadir mi interés personal en aprender el lenguaje C++, que hasta el momento de la realización del Trabajo Fin de Grado no había utilizado. Otro factor clave es el interés por obtener una visión global de la situación actual y las tecnologías usadas en la computación paralela, a lo que se puede añadir la utilización de una de las tecnologías más novedosas e innovadoras hasta la fecha, como es Array Building Blocks.

1.2.Objetivos

Este trabajo tiene como objetivo principal la aplicación del modelo Intel Array Building Blocks sobre algunas aplicaciones del benchmark PARSEC en varias arquitecturas de memoria compartida. Con este objetivo se pretende conocer el comportamiento del nuevo modelo de Intel tras medir los resultados en las aplicaciones paralelizadas. Así mismo, se plantean otra serie de objetivos secundarios:

- Evaluar las distintas aplicaciones de la suite del benchmark PARSEC con el fin de descubrir la utilidad de un benchmark y escoger las aplicaciones más afines a ArBB.
- Comparar las aplicaciones paralelizadas con Intel ArBB con otros modelos de programación paralela como OpenMP o Intel TBB y, de este modo, determinar si posee un rendimiento mejor o peor en comparación con otras tecnologías.
- Mostrar una visión global del paralelismo en arquitecturas de memoria compartida, haciendo especial hincapié en los distintos tipos de acceso a la memoria de un computador y a las tecnologías de Intel, ArBB y TBB. Gracias a esto, se pretende conseguir el aprendizaje del lenguaje de programación C++ y de algunos de los modelos de programación paralela más relevantes.
- Alcanzar la mayor aceleración posible para las aplicaciones paralelizas en cualquier tipo de arquitectura de memoria compartida, garantizando los mismos resultados que en la versión secuencial.

1.3.Estructura del documento

Dentro de este punto, se listan los distintos capítulos que componen el documento, así como una descripción del contenido:

- **Capítulo 1, Introducción:** Determina la motivación por la que se ha realizado este Trabajo Fin de Grado, los objetivos que se pretenden conseguir con la consecución del mismo y una lista con los acrónimos principales.
- **Capítulo 2, Estado del Arte:** Describe la base teórica sobre la que se sustenta el trabajo realizado, haciendo empeño en las tecnologías y aplicaciones utilizadas.
- **Capítulo 3, Planteamiento del problema:** Lista y detalla los requisitos del trabajo a realizar y las restricciones en el diseño de la solución y su implementación.
- **Capítulo 4, Diseño de la solución:** Describe el proceso realizado hasta llegar a los objetivos planteados y cumplir los requisitos impuestos.
- **Capítulo 5, Evaluación del rendimiento:** Especifica el hardware utilizado, la metodología para la medición de tiempos y los resultados finales.

- **Capítulo 6, Conclusiones:** Contiene un resumen que muestra el cumplimiento de los requisitos, las posibles líneas de investigación posteriores a este trabajo y una opinión personal del trabajo realizado.
- **Capítulo 7, Planificación del trabajo:** Detalla el calendario de dedicación al Trabajo Fin de Grado y las modificaciones sufridas.
- **Capítulo 8, Presupuesto:** Especifica el cálculo del coste global de la realización del Trabajo Fin de Grado.

1.4. Acrónimos

Término	Descripción
API	Application Programming Interface
ArBB	Array Building Blocks
AVC	Advanced Video Coding
AVX	Advanced Vector Extensions
C++ AMP	C++ Accelerated Massive Parallelism
CUDA	Compute Unified Device Architecture
DLP	Data-Level Parallelism
GPU	Graphics processing unit
IDE	Integrated Development Environment
ILP	Instruction-Level Parallelism
MIT	Massachusetts Institute of Technology
NPTL	Native POSIX Thread Library
NUMA	Non-Uniform Memory Access
OpenMP	Open Multiprocessing
PARSEC	Princeton Application Repository for Shared-Memory Computers
Pthreads	POSIX Threads
RAM	Random access memory
RMS	Recognition, Mining and Synthesis
SIMD	Single Instruction Multiple Data
SO	System operative
SPH	Smoothed-particle hydrodynamics
TBB	Threading Building Blocks
TLP	Thread-Level Parallelism

2. ESTADO DEL ARTE

Dentro de este capítulo, se describe la base teórica sobre la que se sustenta el Trabajo Fin de Grado realizado. En este punto es necesario aportar una idea general del propósito de un benchmark así como la composición y utilización del benchmark PARSEC. Puesto que el trabajo realizado se ha centrado en dos aplicaciones, en este capítulo se incorpora una descripción técnica de la funcionalidad de cada algoritmo. El paralelismo en arquitecturas de memoria compartida es el otro protagonista principal, dentro de este punto se incluyen las tecnologías utilizadas e información relevante para la evaluación de los resultados obtenidos.

2.1.Benchmark PARSEC

Un benchmark es una carga de trabajo artificial que incluye las características más importantes de cargas de trabajo reales y relevantes. Generalmente, los benchmarks son aplicaciones pequeñas, eficientes y controlables y su objetivo es evaluar un sistema. El término benchmark también hace referencia a la técnica para evaluar el rendimiento de un computador. Los propósitos de un benchmark son:

- Comparar el rendimiento de dos sistemas para determinar cuál de ellos es mejor.
- Evaluar por separado los distintos elementos de un sistema (procesador, memoria RAM, GPU, disco duro, SO, consumo de energía, etc.)
- Comprobar las limitaciones de una computadora en aras de mejorar su rendimiento.

El uso de este tipo de mediciones es muy útil dentro del campo de la informática tanto desde punto de vista técnico como del comercial, debido a esto hay una gran variedad de suites de benchmarks disponibles en el mercado. Algunos de los benchmarks más conocidos son la familia SPEC, 3DMark o Ciusbet.

La suite benchmark PARSEC (*Princeton Application Repository for Shared-Memory Computers*)[2] posee un conjunto de trece programas con el objetivo de evaluar procesadores multicore en arquitecturas de memoria compartida. La primera versión de esta suite fue publicada en el año 2008 y en ella participaron instituciones como la universidad de Princeton, Intel o el MIT. Todos los programas incluidos en la suite han sido previamente paralelizados y se han desarrollado con fines académicos. Una de las cualidades diferenciadores del benchmark PARSEC es la elección de los programas, caracterizados por su innovación y diversidad. Muchas de estas aplicaciones, pertenecen al repositorio de Intel RMS[3] destaca el crecimiento emergente de aplicaciones de propósito general en la computación de alto rendimiento. En la Tabla 1 se presenta un resumen con las características principales de los programas donde se aprecia la diversidad anteriormente descrita.

Programa	Dominio	Paralelización		Carga de trabajo	Uso de datos	
		Modelo	Granularidad		Compartición	Intercambio
Blackscholes	Análisis financiero	Paralelización de datos	Grueso	Pequeño	Bajo	Bajo
Bodytrack	Inteligencia artificial	Paralelización de datos	Medio	Medio	Alto	Medio
Canneal	Ingeniería	No estructurado	Fino	Ilimitado	Alto	Alto
Dedup	Almacenamiento corporativo	Flujo de datos	Medio	Ilimitado	Alto	Alto
Facesim	Animación	Paralelización de datos	Grueso	Grande	Bajo	Medio
Ferret	Búsqueda de similitudes	Flujo de datos	Medio	Ilimitado	Alto	Alto
Fluidanimate	Animación	Paralelización de datos	Fino	Grande	Bajo	Medio
Freqmine	Minería de datos	Paralelización de datos	Medio	Ilimitado	Alto	Medio
StreamCluster	Minería de datos	Paralelización de datos	Medio	Medio	Bajo	Medio
Swaptions	Análisis financiero	Paralelización de datos	Grueso	Medio	Bajo	Bajo
Vips	Procesado multimedia	Paralelización de datos	Grueso	Medio	Bajo	Medio
X264	Procesado multimedia	Flujo de datos	Grueso	Medio	Alto	Alto

Tabla 1. Aplicaciones incluidas en el benchmark PARSEC 1.0

El benchmark PARSEC proporciona seis entradas de distintos tamaños para cada aplicación que permiten comprobar el correcto funcionamiento de esta, obtener resultados preliminares y finalmente una evaluación completa. Estas son las seis entradas proporcionadas:

- **test:** entrada muy pequeña utilizada para probar la funcionalidad básica del programa.
- **simdev:** entrada pequeña que permite aproximarse al comportamiento real de la aplicación. Suele ser usado para tareas de desarrollo.
- **simsmall, simmedium y simlarge:** entradas de diversos tamaños apropiadas para evaluar el rendimiento en microarquitecturas.
- **native:** entrada de gran tamaño que simula una entrada real.

A continuación se describen las trece aplicaciones (Tabla 2) que componen el benchmark PARSEC:

- **Bodytrack**: Esta aplicación, de inteligencia artificial, realiza el seguimiento a un cuerpo 3D, a través de varias cámaras y mediante una secuencia de imágenes. Este programa fue incluido debido a la creciente importancia de los algoritmos de inteligencia artificial en áreas tales como la vigilancia por video, animación de personajes e interfaces.
- **Canneal**: El *kernel* Canneal utiliza un modelo de caché específica (*cache-aware*) para la algoritmo de *recocido simulado*. Esta técnica es usada para la búsqueda meta-

heurística del menor coste de enrutamiento en el diseño de un chip. Es una de las pocas aplicaciones de la suite que permiten una paralelización granular fina.

- **Dedup**: Este *kernel* se encarga de comprimir un flujo de datos mediante la técnica denominada *deduplicación*. Con el fin de eliminar redundancia en la información, esta técnica combina una compresión global y local de los datos. La razón por el que ha sido incluido en esta suite se debe a la importancia que está cobrando está técnica en la nueva generación de sistemas de almacenamiento de copias de seguridad.
- **Facesim**: Es una aplicación que genera una animación realista de una cara modelada mediante la simulación de la física fundamental. El algoritmo de Facesim es muy utilizado para generar efectos de animación más realistas.
- **Ferret**: Dicha aplicación proporciona un motor de búsqueda para la obtención de resultados similares a partir de contenido multimedia. Esta aplicación ha sido configurada para realizar búsquedas de imágenes, ya que es una de las opciones computacionalmente más costosas. Ferret es útil por sus prestaciones y por el modelo de paralelización con el que ha sido implementado.
- **Freqmine**: Aplicación que cuenta con una versión basada en el patrón del crecimiento frecuente y que ejecuta el método de minería denominado conjunto de elementos frecuentes. Su inclusión se debe al uso creciente de técnicas de minería de datos en la actualidad.
- **Raytrace**: Esta aplicación se encarga de realizar el cálculo del trazado de rayos que normalmente se emplean para las animaciones en tiempo real, como por ejemplo los juegos para ordenador. Esta aplicación fue incluida en la segunda versión de la suite.
- **Streamcluster**: El algoritmo de Streamcluster resuelve el problema en línea del algoritmo de agrupamiento. Al igual que Freqmine, Streamcluster fue incluido debido a la importancia de algoritmos de minería de datos en la actualidad.
- **Swaptions**: Aplicación del ámbito financiero que utiliza el *framework Heath–Jarrow–Morton* basado en el modelo de Monte Carlo, el cual se encarga de computar los precios de un conjunto de derivados financieros denominados *swaptions*. El método de Monte Carlo es uno de los métodos no determinísticos más importante y su inclusión como benchmark es de carácter obligado.
- **Vips**: Esta aplicación se basa en el Sistema de Procesamiento de Imagen VASARI (VIPS) que se encarga del procesado de imágenes de gran tamaño. Su algoritmo es muy complejo y por ello la suite PARSEC sólo incluye las operaciones de transformación afín y convolución.
- **X264**: Codificador de video H.264/AVC (*Advanced Video Coding*). H.264 describe la compresión con pérdida de un flujo de vídeo y también es parte de la norma ISO / IEC MPEG-4. La flexibilidad y la amplia gama de aplicación de la norma H.264 y su ubicuidad en los sistemas de vídeo de última generación son las razones de la inclusión de x264 en el conjunto de aplicaciones de PARSEC.

Las aplicaciones Blackscholes y Fluidanimate son descritas con detalle a continuación. Debido al gran número de programas y frente a la imposibilidad de abarcarlos todos,

mi tutor y yo decidimos trabajar sobre estas dos aplicaciones. El principal criterio a tener en cuenta en el proceso de selección ha sido la adaptabilidad al modelo de Intel Array Building Blocks.

Aplicación	Modelo de paralelización			Portabilidad								
				Compilador			SO			CPU		
	Pthreads	OpenMP	Intel TBB	gcc 4.2	gcc 4.3	Icc 10.1	Linux	Solaris 10	Windows	I386	X86_64	Sparc
Blackscholes	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Bodytrack	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Canneal	Sí	No	No	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Dedup	Sí	No	No	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Facesim	Sí	No	No	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Ferret	Sí	No	No	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Fluidanimate	Sí	No	Sí	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Freqmine	No	Sí	No	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Raytrace	Sí	No	No	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
StreamCluster	Sí	No	Sí	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Swaptions	Sí	No	Sí	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí
Vips	Sí	No	No	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
X264	Sí	No	No	Sí	Sí	Sí	Sí	Sí	No	Sí	Sí	Sí

Tabla 2. Listado de las características de las aplicaciones del PARSEC [4]

2.1.1. Blackscholes

Blackscholes en el benchmark PARSEC

En líneas generales Blackscholes es interesante por la importancia del problema que resuelve y la facilidad para ser paralelizada, por ello forma parte del repositorio de Intel RMS y del benchmark PARSEC. Esta aplicación es una de las más extendidas, permitiendo ser ejecutada en sistemas operativos Linux, Windows y Solaris 10. Soporta arquitecturas i386, Sparc y x86_64. La suite incluye las versiones paralelizadas utilizando los modelos Pthreads, OpenMP e Intel TBB.

Introducción

Blackscholes[5] es utilizado en el ámbito del análisis financiero, la funcionalidad de esta aplicación se basa en la resolución del modelo Black-Scholes, ganador del premio Nobel de Economía en 1997 y pudiéndose catalogar como la aportación más relevante en el campo de la teoría y práctica financiera de los últimos años. Esta ecuación es utilizada ampliamente en los mercados financieros permitiendo, bajo unos supuestos, estimar el precio de una opción Europea en una fecha futura "sin riesgos". La ecuación

permite las opciones de compra (*Call*) y de venta (*Put*). Las opciones financieras son instrumentos que otorgan al comprador o vendedor el derecho de realizar una transacción a un precio fijado y en una fecha determinada.

Estructura de la información y algoritmo

Esta aplicación calcula el precio estimado de una cartera de opciones, procesando cada una de forma independiente. Los parámetros necesarios para resolver la ecuación Blacks-Scholes son:

- Tipo de operación, pudiendo ser una opción de compra o venta. (P o C)
- Precio del activo en el momento de la valoración. (S)
- Precio predeterminado para esa opción. (K)
- Tasa de interés en tiempo continúo. (R)
- Volatilidad del precio. (σ)
- Tiempo restante hasta el vencimiento de la opción. (T)
- Función de distribución acumulativa de la distribución normal. (N)

Siendo la ecuación para la compra de opciones:

$$C(S,T) = N(d1)S - N(d2)Ke^{-rT}$$

Dónde:

$$d1 = \frac{ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d2 = \frac{ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} = d1 - \sigma\sqrt{T}$$

Y la ecuación para la venta:

$$P(S,t) = Ke^{-rT} - S + C(S,T) = N(-d2)Ke^{-rT} - N(-d1)S$$

La aplicación Blackscholes realiza la lectura de los parámetros anteriormente mencionados para cada opción. Una vez completada la cartera de opciones se calcula el precio de cada una por separado haciendo uso de las ecuaciones descritas. Finalmente se escriben los precios calculados en un fichero de texto que será la salida de la aplicación.

2.1.2. Fluidanimate

Fluidanimate en el benchmark PARSEC

Fluidanimate es una aplicación que forma parte de Intel RMS y que fue incluida en la primera versión de la suite del benchmark PARSEC. Esta aplicación puede ejecutarse en sistemas operativos Linux o Solaris 10 y soporta arquitecturas i386, x86-64 y Sparc. El propio benchmark PARSEC incorpora la versión secuencial de Fluidanimate y las versiones paralelizadas con el modelo Intel TBB y Pthreads, en la próxima versión el benchmark se incluirá la versión paralelizada con OpenMP[1].

Introducción

Esta aplicación se encarga de la simulación de fluidos utilizando el método computacional *Smoothed Particle Hydrodynamics* (SPH) con el fin de resolver las ecuaciones de *Navier-Stokes*. Estas ecuaciones son un conjunto de derivadas parciales no lineales que describen el movimiento de un fluido newtoniano. Los fluidos newtonianos[6] son aquellos en los que su viscosidad es independiente del movimiento del mismo, un claro ejemplo de fluido newtoniano es el agua. Dichos fluidos están muy presentes en la naturaleza y ciencias como la oceanografía o la astrofísica utilizan el método SPH para el cálculo del flujo de estos fluidos.

Estructura de la información

Dentro de la aplicación Fluidanimate, para la simulación de un fluido, el líquido es dividido en celdas que sirven como contenedores de partículas. Cada celda puede albergar un máximo de dieciséis partículas (Ilustración 2).

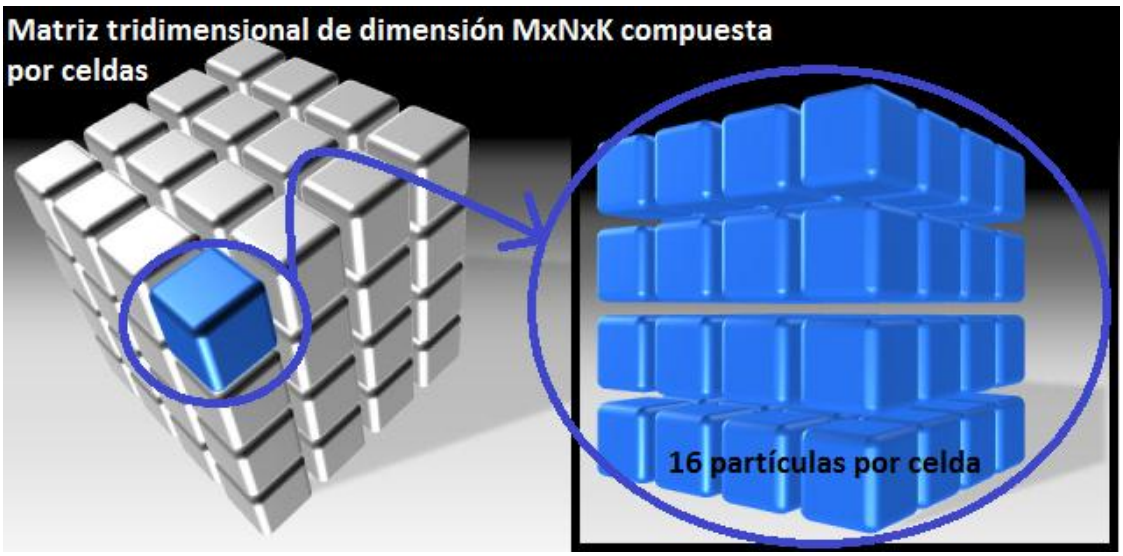


Ilustración 2. Estructura de celdas en Fluidanimate

[1] https://lists.cs.princeton.edu/pipermail/parsec-users/2012-January/001274.html

Cada partícula posee los siguientes atributos:

- **Aceleración**: Vector tridimensional que expresa el cambio de velocidad de la partícula por unidad de tiempo.
- **Densidad:** Relación entre la cantidad de masa de la partícula y su volumen.
- **Posición:** Vector tridimensional que describe la localización de la partícula en el espacio.
- **Velocidad:** Vector tridimensional que muestra la distancia recorrida de la partícula por unidad de tiempo.
- **Viscosidad:** Vector tridimensional que expresa la oposición de la partícula a las deformaciones tangenciales.

Fluidanimate posee una clase denominada *Vec3* que simula un vector tridimensional. Esta clase es usada para representar los vectores de la aceleración, posición, velocidad y viscosidad de cada partícula. *Vec3* utiliza números en coma flotante (*floats*) para representar las magnitudes de los ejes x, y, z. *Vec3* incorpora operaciones vectoriales como el cálculo del producto escalar o la multiplicación de vectores. Gracias a esta clase se consigue la representación de los atributos de las partículas. A nivel de celda, Fluidanimate incorpora una estructura denominada *Cell* compuesta por cuatro vectores con objetos de la clase *Vec3* y un vector unidimensional que representa la densidad, la tamaño de los cinco vectores es de dieciséis elementos que se corresponde con el número máximo de partículas contenidas por celda. Con la fusión de la clase *Vec3* y la estructura *Cell* ya se consigue representar una celda, un vector compuesto de estructuras de tipo *Cell* permite construir la matriz tridimensional, mostrada en la Ilustración 2, donde se realizará la simulación del fluido.

Algoritmo

El primer paso que se realiza en la ejecución de Fluidanimate es la lectura de la posición, velocidad y viscosidad de las partículas a partir de un fichero de entrada. El fichero de entrada determina el tamaño de la matriz tridimensional y el número de partículas que tendrá. Las partículas son introducidas en las diferentes celdas en función de la posición que ocupan, pudiéndose dar el caso de desechar partículas al llenarse una celda. Esta función se denomina *InitSim()* y es llamada una única vez por cada ejecución.

A partir de la primera matriz tridimensional construida se aplica un filtro a cada partícula modificando su posición. Si la nueva posición es diferente a la anterior implicará la recolocación de la partícula en una nueva celda o la eliminación de esta si supera los límites establecidos en la matriz. Esta función se denomina *RebuildGrid()*.

Tras construir la matriz final, se proporciona el valor por defecto de la densidad y la aceleración a todas las partículas existentes. Acto seguido, comienza la interacción entre partículas. El algoritmo es similar a un tipo bien conocido llamado *Stencil*, este tipo de operación es explicado con detalle en el punto 2.1.2.1. A modo de resumen, el proceso es el siguiente: para cada celda se buscan las celdas adyacentes a esta, una vez localizadas todas las celdas vecinas, comienza la interacción a nivel de partícula,

pudiéndose dar un máximo de 431[2] interacciones para cada partícula. El resultado de esta operación es la modificación de la densidad de cada partícula, para ello se tiene en cuenta las posiciones de las partículas de su propia celda y de las partículas contenidas en las celdas vecinas.

Después del cálculo de densidades, el siguiente paso a dar es la modificación de los valores calculados mediante unas constantes predefinidas. En este momento se repite la interacción entre partículas descrita en el párrafo anterior, con la diferencia de que en este caso el objetivo es calcular el valor de la aceleración. Para ello es necesario evaluar en cada interacción la densidad, la posición y la velocidad de las partículas protagonistas además de agentes externos como la aceleración exterior. Este conjunto de operaciones están englobadas dentro de la función *ComputeForces()*.

Una vez calculadas las densidades y las aceleraciones de todas las partículas se simula la nueva posición de las partículas y las posibles colisiones, ya sea con otra partícula o con el límite de la matriz tridimensional. Esta función, denominada *ProcessCollisions()*, no elimina una partícula que haya colisionado, se limita a acelerarla o decelerarla.

Finalmente, se actualiza la posición, velocidad y viscosidad a partir de la aceleración final y el tiempo transcurrido. Esta función hace uso del algoritmo *Leap-Frog*, que se trata de una variante del algoritmo de dinámica molecular *Verlet*. Una vez terminado este paso, se repetirá el proceso en función del número de iteraciones que el usuario introdujo por argumento. Cada iteración supone un incremento en el paso del tiempo de una unidad de tiempo de 0,005 segundos, este atributo se conoce como *timestep*. Esta función es identificada por el nombre de *AdvanceParticles()*.

RebuildGrid(), ComputeForces, ProcessCollisions() y *AdvanceParticles* son ejecutadas a través de la función *AdvanceFrame()* que se encarga de ejecutar las sucesivas iteraciones del programa. Queda por mencionar la función *SaveFile()*, de carácter opcional, que se encarga de guardar en un fichero de texto binario los atributos resultantes de las partículas computadas tras la simulación.

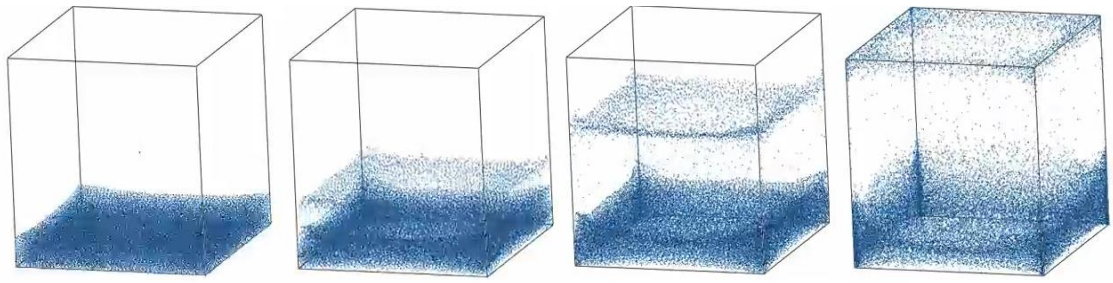


Ilustración 3. Capturas de una animación de fluidos en Fluidanimate [7]

[2] 27 celdas vecinas (incluida la suya propia) * 16 partículas – 1 (partícula anfitriona)

2.1.2.1. Operación *Stencil*

Las operaciones de tipo *Stencil* son muy comunes en el mundo de la computación y poseen un amplio rango de aplicaciones en los campos de la ingeniería y de la ciencia, como por ejemplo en la computación dinámica de fluidos. Las operaciones computacionales *Stencil*[8] son usadas para resolver ecuaciones diferenciales parciales, como podría ser la de Navier-Stokes utilizada en Fluidanimate. Estas operaciones realizan, mediante un bucle, un gran número de barridos sobre un vector multidimensional en el que el valor de cada elemento es repetidamente modificado basándose en los valores de los elementos vecinos.

Hay una gran diversidad de *Stencils*[9] estas son las características básicas que pueden ser utilizadas para definirlos:

- **Dimensión**: Se corresponde con el número de ejes que posea el vector sobre el que se realizará la operación. En la Ilustración 4 se puede observar un *Stencil* de tres dimensiones (a) y un *Stencil* de dos dimensiones (b).
- **Huella:** Expresa el menor volumen que encierra a todos los elementos del vector multidimensional involucrados en la interacción.
- **Orden:** Este parámetro especifica la distancia máxima a la que se encuentran los vecinos. Un *Stencil* de orden uno se correspondería con los vecinos estrictamente adyacentes.
- **Tamaño:** Característica que muestra el número total de elementos involucrados dentro del cómputo de cada elemento.

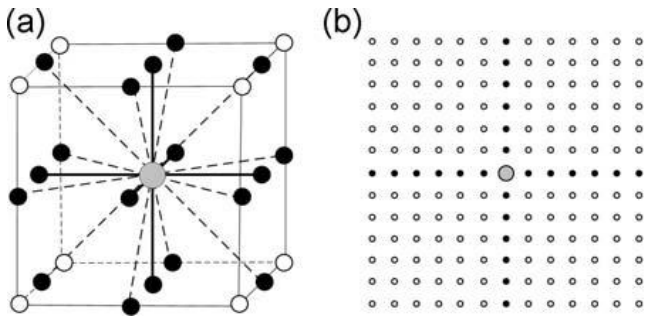


Ilustración 4. Dimensiones de los *Stencils*

Otro factor importante es identificar cómo será la operación de actualización y el orden a seguir. Si el recorrido de la matriz no influye en la obtención de resultados habrá más libertad en el algoritmo y se permitirá una mejora de este. Dentro de la operación de actualización hay dos tipos:

- **Jacobi**: Las operaciones de lectura y escritura se realizan en matrices diferentes, lo que evita problemas de sincronización. Este tipo se corresponde con la imagen (a) de la Ilustración 5.
- **Gauss-Seidel:** La actualización se realiza en la propia matriz. Este tipo se corresponde con la imagen (b) de la Ilustración 5.

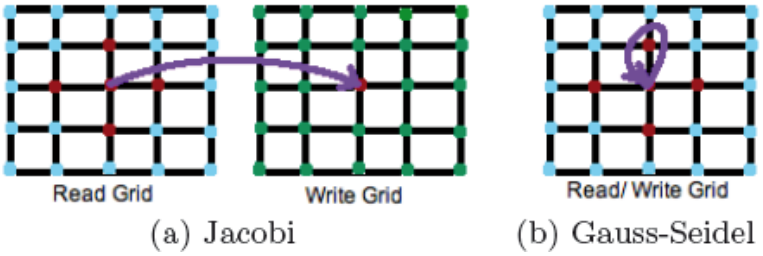


Ilustración 5. Tipos de actualización en los *Stencils*

Además de estas características, las operaciones *Stencils* pueden ser diferenciadas por el tipo de límites (circulares o fijos) o por la aplicación de coeficientes en función de la lejanía del elemento vecino (constantes o variables).

Centrándose en la aplicación Fluidanimate y más concretamente en la función *ComputeForces()*, su algoritmo se asemeja a dos *Stencils* de 27 puntos (**¡Error! No se ncuentra el origen de la referencia.**). Siguiendo las propiedades anteriormente descritas, este *Stencil* cumple con las siguientes características:

- **Dimensión:** 3
- **Huella:** 3^3
- **Orden:** 1
- **Tamaño:** 27
- **Tipo de actualización:** Jacobi
- **Límites:** Fijos
- **Coeficientes**: Constantes

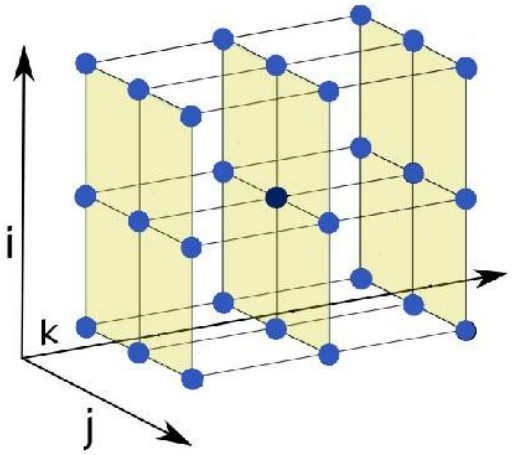


Ilustración 6. Stencil de 27 puntos en Fluidanimate

No obstante, el algoritmo implementado en Fluidanimate no deja de ser una aproximación a un *Stencil*. La principal diferencia radica en que mientras el acceso a la información se hace a nivel de partículas, la búsqueda de vecinos se realiza a nivel de celdas. Así pues, el *Stencil* de 27 puntos es válido para determinar la búsqueda de vecinos, pero no para actualizar los valores de las partículas. Esto implica que el acceso a la información no es uniforme y, por tanto, no pueda ser predeterminado. La utilización de distintas magnitudes para la búsqueda de vecinos y la lectura de la información y modificación implica que la distancia a una partícula vecina desde dos partículas contenidas dentro de la misma celda es distinta.

2.2.Paralelismo en arquitecturas de memoria compartida

El paralelismo[10] es una técnica de programación que consiste en dividir un problema computacional y resolverlo de forma concurrente. Gracias a esta técnica, se consigue mejorar el rendimiento de las aplicaciones, reduciendo su tiempo de cómputo. Dentro del paralelismo se pueden distinguir varias clases: paralelismo a nivel de bits (a mayor ancho de palabra menor número de instrucciones), paralelismo a nivel de instrucción (ILP), paralelismo a nivel de tareas (TLP) y paralelismo a nivel de datos (DLP). Mientras que ILP optimiza las aplicaciones en un único procesador, consiguiendo ejecutar varias instrucciones a la vez, TLP y DLP se encargan de dividir la carga de cómputo sobre los distintos nodos de la computadora. Las diferencias entre TLP y DLP se basan en las operaciones a ejecutar, por un lado TLP asigna distintas secuencias de operaciones a cada procesador, por el otro lado DLP replica la misma operación a todos los nodos de cómputo distribuyendo los datos. El paralelismo a nivel de datos se consigue gracias a la arquitectura SIMD (Ilustración 7) que propuso Flynn en el año 1972. También, la arquitectura SIMD (*Single Intruction Multiple Data)* permite ejecutar una única operación sobre varios datos de forma simultánea mediante la utilización de un procesador que posea unidades vectoriales las cuales proporcionan un nivel adicional de paralelismo.

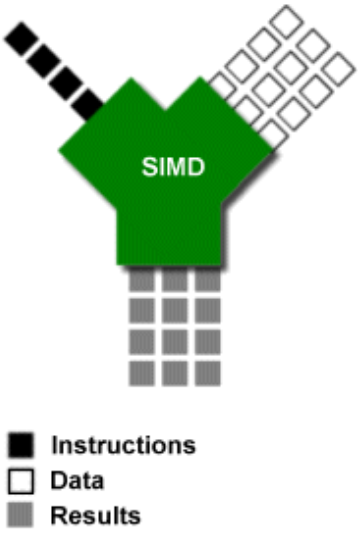


Ilustración 7. Arquitectura SIMD

Por estos motivos, las especificaciones de una computadora son muy importantes en la paralelización de una aplicación. Centrándose en el procesador, el número de cores delimitará el número de hilos que se pueden crear en un instante dado, lo que permitirá una mayor división de la carga de trabajo. No obstante, un mayor número de cores no tiene porqué producir un mayor rendimiento, ya que la frecuencia de estos es un dato a tener muy en cuenta, cuanto mayor sea la frecuencia mayor será el número de instrucciones que pueden ejecutar en una unidad de tiempo. Para finalizar, las características de los procesadores han evolucionado y ahora cuentan con el uso de

técnicas multi-hilo en un mismo núcleo, cómo la conocida hyper-threading de los procesadores de Intel. Además, la utilización de la vectorización provoca grandes reducciones en el tiempo de cómputo, por lo que contar con juegos de instrucciones vectoriales permitirá un aumento de la paralelización. Un ejemplo de instrucciones vectoriales es AVX que fue incorporado por Intel en los procesadores Sandy Bridge. Algunas bibliotecas de paralelización como Array Building Blocks las utilizan explotando sus ventajas.

Las arquitecturas con memoria compartida[11] son aquellas en las que todos los procesadores comparten el mismo rango de direcciones en memoria. Dentro de los multiprocesadores aparecen dos tipos de diseños para el acceso a memoria: memorias de acceso uniforme (UMA) y las memorias de acceso no uniforme (NUMA). El diseño UMA (Ilustración 8) permite que todos los procesadores puedan acceder a las distintas posiciones de memoria invirtiendo el mismo tiempo de acceso, no obstante, los procesadores pueden disponer de una memoria caché privada.

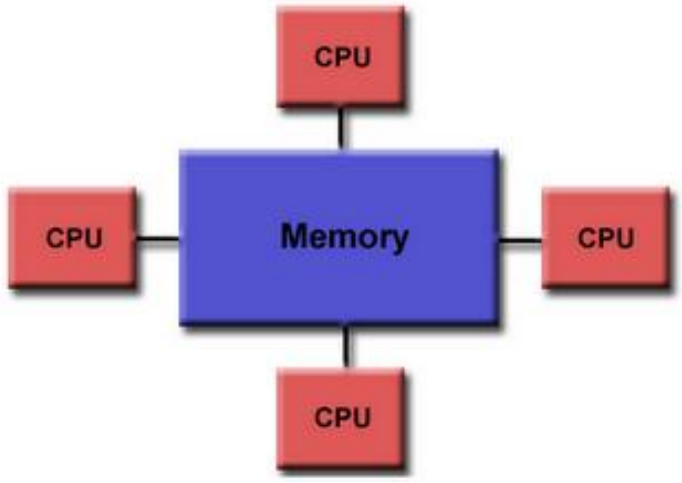


Ilustración 8. Arquitectura UMA

En las arquitecturas NUMA el tiempo de acceso a un dato en memoria depende del procesador que solicite la petición. El procesador puede acceder a su memoria local más rápido a que a la memoria local de otro procesador. La principal ventaja de esta arquitectura es la escalabilidad, ya que existe un bus que interconecta los distintos procesadores permitiendo que puedan añadirse nuevos nodos sin problemas. Por lo general, este modelo es usado en arquitecturas multi-socket en las que coexisten varios procesadores independientes. La mayor desventaja es que, al no compartir los bancos de memoria, surge la necesidad de incluir un mecanismo que se encargue de la sincronización y coherencia de las distintas memorias, lo que implica que los fallos de caché de un proceso ejecutado en varios procesadores simultáneamente añadan una gran penalización en el tiempo de cómputo.

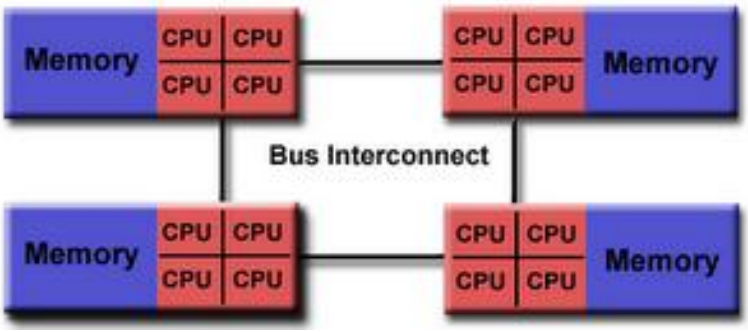


Ilustración 9. Arquitectura NUMA

La memoria caché que tiene cada procesador permite reducir considerablemente el tiempo de acceso a la información más relevante durante la ejecución de un proceso. Teniendo en cuenta el tamaño de la caché del procesador y conociendo las principios básicos de localidad espacial y temporal es posible reducir el número de fallos de caché y por ende el tiempo de ejecución del programa. Un ejemplo sencillo es el de una matriz en el que sus elementos están almacenados en memoria por fila, si un proceso la recorriese por columnas generaría un gran número de fallos de memoria debido a la no utilización del principio de localidad espacial.

Una vez conocidos estos factores, es tarea del ingeniero decidir si un determinado código puede ser paralelizado en una computadora específica, esta no es una tarea sistemática, ya que depende de muchos factores: tipos de variables utilizadas, utilización de operaciones aritméticas en coma flotante, bucles, acceso a memoria, número de procesadores, división del trabajo, etc.

La posterior evaluación de los resultados obtenidos es otras de las tareas que forman parte de la paralelización de una aplicación. Para medir el rendimiento de un determinado código se toman los tiempos empleados en la ejecución de la versión paralela y en la versión secuencial de la aplicación y se utiliza la siguiente formula:

$$Aceleración = \frac{\text{Tiempo versión secuencial}}{Tiempo\ versión\ paralelizada}$$

La aceleración resultante mostrará el número de veces que la aplicación paraleliza es mejor respecto la secuencial. Esta fórmula también puede ser útil para comparar distintas versiones paralelizadas con distintos modelos y comparar sus rendimientos. No obstante, el resultado de la formula no tiene en cuenta el número de procesadores empleados, información relevante si se quiere conocer cuál es la aceleración que conseguimos por cada procesador empleado, es decir, la eficiencia de los procesadores. Para ello puede utilizarse la siguiente formula:

$$Aceleración\ por\ procesador = \frac{Aceleración}{Nº\ de\ procesadores}$$

Finalmente, la ley de Amdahl es útil para determinar la aceleración máxima que se puede lograr al paralelizar un programa en una arquitectura multicore. Para ello hay que tener en cuenta la parte secuencial que no se modificó y el número de procesadores empleados. El resultado puede ser tomado como el límite ideal de la aceleración que se puede conseguir con la paralelización de un determinado código. No obstante, y como se verá en la evaluación de los resultados, la ley de Amdahl no tiene en cuenta el paralelismo que se realiza mediante la vectorización y que modelos de programación paralela como ArBB si tienen en cuenta. La fórmula para calcular esta aceleración máxima es la siguiente:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Donde S es la aceleración resultante, P la proporción de tiempo que ocupa la parte paralelizada y N el número de procesadores empleados.

2.3.Modelos de programación paralela

2.3.1. Pthreads

POSIX threads (Pthreads)[12] es la especificación de una API que permite la programación paralela en arquitecturas multicore de memoria compartida. Esta especificación fue estandarizada por primera vez en el IEEE POSIX 1003.1c en el año 1995. Actualmente, es soportado por sistemas operativos UNIX, Linux y Windows. La implementación de Pthreads más popular y extendida es NPTL (*Native POSIX Thread Library*) que fue incluida en el *kernel* de Linux a partir del año 2003.

Esta interfaz de programación para el lenguaje C hace uso de los denominados threads para proporcionar el paralelismo a nivel de hilos descrito en el punto anterior. Mientras que un proceso es cualquier programa en ejecución independiente del resto de procesos, un hilo es un subproceso que comparte memoria con otros hilos. No obstante, cada hilo tiene a su disposición el contador de programa, la pila de ejecución, el valor de los registros y su propio estado. Al ser compartida el resto de la información, es necesario incluir mecanismos que permitan la sincronización deseada y evitar condiciones de carrera. Para ello se incluyen mecanismos de bloqueo y exclusión mutua como los conocidos *mutex* o semáforos.

Dentro de las implementaciones de los hilos hay dos categorías, los ULT (*user level thread*) y KLT (*kernel level thread*). En los hilos a nivel de usuario, el trabajo de creación y gestión de hilos lo gestiona la propia aplicación, esto implica que el núcleo del *kernel* no

sea consciente de la utilización de varios hilos. La otra opción son los hilos a nivel de *kernel*, donde el *kernel* puede planificar simultáneamente múltiples hilos del mismo proceso en múltiples procesadores. Pese a que NPTL soporta las dos tipos de hilos, los hilos a nivel de usuario no tienen trascendencia en la programación paralela en arquitecturas multiprocesadores debido a que toda la creación de hilos se realiza en un único procesador.

Pthreads ofrece un gran acercamiento al hardware lo que permite tener un mayor control en la gestión de hilos y un mejor aprovechamiento de las características de cada computador, de lo que se deduce una gran reducción en el tiempo de cómputo. No obstante, este mayor acercamiento al hardware supone una baja abstracción que provoca que el programador tenga que tener en cuenta los problemas acarreados por la compartición de información entre hilos, la distribución de la carga computacional, excesivos cambios de contexto o la portabilidad debido a la gran dependencia del computador utilizado. Las diversas utilidades que pueden darse a la interfaz de Pthreads ha supuesto que su uso esté muy extendido, por el contrario, al no estar orientado sobre un campo específico de la computación dificulta la obtención de un rendimiento óptimo en el paralelismo a nivel de hilos.

2.3.2. OpenMP

El estándar Open Multi-Processing (OpenMP)[13] , lanzado en 1997, especifica una API que expresa paralelismo a nivel de hilos (TLP) para arquitecturas de memoria compartida. Esta paralelización se consigue mediante la utilización de directivas del compilador y llamadas a subrutinas. OpenMP soporta varios sistemas operativos y está disponible para los lenguajes C, C++ y Fortran. Gracias a la utilización de directivas dentro del código secuencial se evita la modificación masiva del código a diferencia de Pthreads.

OpenMP utiliza una estrategia *fork-join*. Este modelo comienza con un único hilo de ejecución, conocido como el hilo maestro, hasta llegar a una directiva, en ese momento la carga de trabajo se divide para que la región paralela sea ejecutada por el conjunto de hilos. Cuando la fase a paralelizar termina el hilo maestro destruye el resto de hilos y continúa la ejecución secuencial.

Inicialmente OpenMP se orientó a la paralelización de grandes bucles regulares, no obstante, la nueva versión del estándar ha permitido la incorporación de paralelismo orientado a tareas similar al de Intel TBB. Este modelo de programación paralela no otorga un control tan exhaustivo sobre el hardware como Pthreads, lo que evita tener en cuenta las complicaciones derivadas de la gestión de los hilos y por ende, permite una mayor portabilidad. El no poder establecer afinidad entre los núcleos del procesador y los hilos y el no explotar el principio de localidad de los datos en memoria hace que OpenMP no sea muy eficiente en arquitecturas NUMA. Otra desventaja de esta biblioteca es la dependencia a arquitecturas de memoria compartida

ya que no incorpora la funcionalidad necesaria para la paralelización a nivel de hilos en sistemas distribuidos.

2.3.3. Threading Building Blocks

Intel Threading Building Blocks (TBB)[14] es un modelo de programación paralela a nivel de tareas (TLP) basado en plantillas, tipos de datos y algoritmos para el lenguaje de programación C++. Su primera versión fue lanzada en el año 2006 con la intención de proveer al programador de un alto nivel de abstracción en la paralelización orientada a tareas, actualmente es soportado en sistemas operativos Linux, Windows y MAC; y es compatible con otros modelos de programación paralela como OpenMP, MPI o ArBB. TBB se centra en la división del trabajo en pequeñas tareas en lugar de los hilos de Pthreads u OpenMP, estas tareas son asignadas de forma automática a los hilos disponibles. Las ventajas de utilizar tareas en lugar de hilos son las siguientes:

- **Balanceo de carga.** Gracias a la utilización de tareas es posible dividir una tarea tantas veces como sea necesario y repartir esta división en los procesadores que estén libres. Por otro lado, TBB incorpora la técnica del robo de tareas que permite que un procesador que haya finalizado todas las tareas de su cola, obtenga tareas de otro procesador que aún no ha finalizado su carga de trabajo. Con esta medida se evita tener recursos inactivos.
- **Manejo de tareas poco costoso.** La creación y destrucción de hilos es una operación costosa. Al contrario que OpenMP, TBB crea todos los hilos disponibles al principio y no los destruye hasta la finalización del programa. Por el contrario, realiza las operaciones de creación y destrucción de tareas, que es una operación menos costosa.
- **Política de planificación eficiente.** El sistema operativo utiliza políticas de planificación, como round-robin, sin tener en cuenta las características de cada tarea. Intel TBB aprovecha la información contenida en la memoria caché y los principios de localidad temporal y espacial de esta para seleccionar el orden de ejecución más eficiente. Esta medida supone un aumento importante de la eficiencia en arquitecturas en las que los fallos de caché supone una gran penalización, como en las arquitecturas NUMA.
- **Alto nivel de abstracción.** Gracias a la utilización de tareas, el diseño de la aplicación paralelizada es más sencillo puesto que se asemeja más a la forma de pensar del programador.

Para la creación y manejo de las tareas es necesario al inicio del programa crear un objeto de tipo *tbb::task_scheduler_init*, donde de forma opcional puede indicarse el número de hilos deseados.

Entre las plantillas de funciones, Intel TBB incorpora un conjunto con los patrones de paralelismo más frecuentes en este tipo de programación. Estas funciones son útiles para evitar errores comunes, simplificar la programación y mejorar la aceleración.

Algunos ejemplos de estas plantillas son los conocidos *parallel_for*, *parallel_reduce* o *parallel_do*. Además de esto, TBB incorpora rangos, para permitir la división recursiva en un espacio de iteraciones, incorpora las clases *filter* y *pipeline* que implementan el patrón de paralelismo pipeline e incorpora contenedores similares a los de la STL de C++ con accesos concurrentes de forma eficiente, tales como *concurrent_hash_map* o *concurrent_queue*.

Respecto a los mecanismos de exclusión mutua que establece Threading Building Blocks, hay una amplia gama de tipos de cerrojos. Pese a que este modelo se caracteriza por la utilización mínima de estos elementos, hay ocasiones en las que es imprescindible su uso. Con el fin de permitir una mejor optimización para cada tipo de situación, Intel TBB permite la utilización de cerrojos escalables como *qeuing_mutex*, recursivos cómo *recursive_mutex* o cerrojos que reducen el tiempo de espera cómo *spin_mutex*.

Debido a las características especiales de cada aplicación es posible que los requisitos no se ajusten a cualquiera de las plantillas incorporadas en Intel TBB. Para solventar esto se permite la utilización del gestor de tareas para trabajar directamente con tareas mediante la clase base *task*. Generalmente, la utilización de tareas permite solucionar problemas más generalistas que a su vez se traduce en un aumento de la complejidad.

Los objetos heredados de la clase *task* son ejecutados a través de la función *execute()*. Dentro de estos objetos es posible especificar la creación de más tareas introduciendo un patrón recursivo. De este modo puede establecerse un grafo de tareas donde los métodos *allocate_root(), allocate_child()* o *allocate_continuation()* permiten especificar la estructura del árbol de tareas. Una vez construido el árbol puede decidirse cuál será el orden en el que se ejecutarán las tareas, para ello se utilizan los métodos *spawn_root_and_wait()*, *spawn()* y *spawn_and_wait_for_all()*. Es tarea del programador decidir cuál será el orden de ejecución. Realizando un recorrido en profundidad se ahorra memoria y se prioriza la ejecución de tareas recién creadas lo que favorece los aciertos de caché, realizando un recorrido en anchura se aumenta el paralelismo potencial, pero implica un aumento del uso de memoria.

Por último, mencionar la posibilidad que ofrece Intel TBB de asignar memoria dinámica de forma concurrente y conseguir un buen alineamiento de la memoria caché con las funciones *scalable_allocator()*, *cache_aligned_allocator()* y *aligned_space()* en lugar de las ya conocidas *malloc()* y *calloc()* de C++. Toda la explicación detallada de Intel TBB y algunos ejemplos interesantes pueden encontrarse en [15].

2.3.4. Array Building Blocks

Intel Array Building Blocks[16] es un modelo de programación paralela para C++ desarrollada por Intel. Su primera versión fue publicada en el año 2010 y aún está considerada en fase experimental. La motivación por la que surgió ArBB se debe a la necesidad de acercar los modelos de programación paralela a los desarrolladores

consiguiendo una gran aceleración, ya que las altas prestaciones de los computadores, a menudo, solo son explotadas por una minoría de programadores experimentados. El objetivo planteado era asemejar el código paralelizado a código secuencial, lo que implica la necesidad de proveer a ArBB de portabilidad, escalabilidad y seguridad implícita. Un claro ejemplo de esto es la relajación del programador al no tener que preocuparse por las especificaciones de cada máquina, puesto que ArBB selecciona de automáticamente el número de procesadores óptimo para cada máquina gracias la generación dinámica del código. Esta característica permite ejecutar el mismo código en distintas máquinas sin la necesidad de modificarlo. La escalabilidad es otras de sus características más importantes, si se incrementa el número de procesadores o mejora la calidad de estos, ArBB se adapta a estas nuevas características. Además de esto ArBB soporta todos los estándares de C++ y es compatible con los compiladores de Intel, GCC y Microsoft Visual C++, por lo que su integración está garantizada.

Centrándose en las características técnicas de esta nueva modelo, ArBB se concentra en la solución de problemas matemáticos de computación intensiva, principalmente recogidos en operaciones vectoriales y matriciales. El alcance principal de esta biblioteca se centra en el paralelismo de datos (DLP) que suele ser más escalable en comparación al paralelismo de tareas. Gracias al uso de la vectorización, ArBB aprovecha los juegos de instrucciones vectoriales incorporados en los procesadores, tales como SSE y AVX. La posibilidad de utilizar estas instrucciones permite que, además de dividir la carga de trabajo de un proceso en varios procesadores, estos a su vez incorporen una mejora del rendimiento. Intel posee varios juegos de instrucciones vectoriales, en 1997 se introdujo MMX que eran instrucciones de 64 bits, a raíz de estas se introdujo en 1999 SSE con un tamaño de 128 bits por instrucción. SSE ha sido mejorada durante sucesivas versiones hasta que finalmente se publicó la versión 4 en el año 2007. Las instrucciones SSE operan utilizan operandos en coma flotante de precisión simple y realizan tareas de diversa índole como transferencia de datos u operaciones aritméticas. Un año más tarde, surge AVX (*Advanced vector extensions*), como un nuevo juego de instrucciones de 256 bits que mejora al anterior SSE. Tanto SSE como AVX son los juegos de instrucciones explota ArBB, para conseguir esto genera durante la fase de compilación instrucciones aptas para este tipo de extensiones.

Array Building Blocks es compilado con un compilador de C++ común como pueda ser gcc. Para conseguir la generación de instrucciones vectoriales, la adaptabilidad a la plataforma y el resto de optimizaciones se realizan dos fases de compilación. Como se puede observar en la Ilustración 10, inicialmente se realiza una compilación con el compilador compatible con C++, una vez realizada esta operación se pasa a enlazar con la librería de ArBB. Durante esta fase es posible depurar la aplicación de manera idéntica a la de un código escrito únicamente en C++, esto permite una mayor simplicidad al programador.

En la segunda fase de la compilación se adapta el código binario a la plataforma utilizada y es ahí cuando se lanza la *JIT (Just in time) Compilation* de ArBB. Este compilador utiliza una máquina virtual para optimizar el código compilado anteriormente permitiendo la vectorización y la utilización de la arquitectura

multicore. Una vez realizado este último paso el código optimizado puede ser ejecutado.

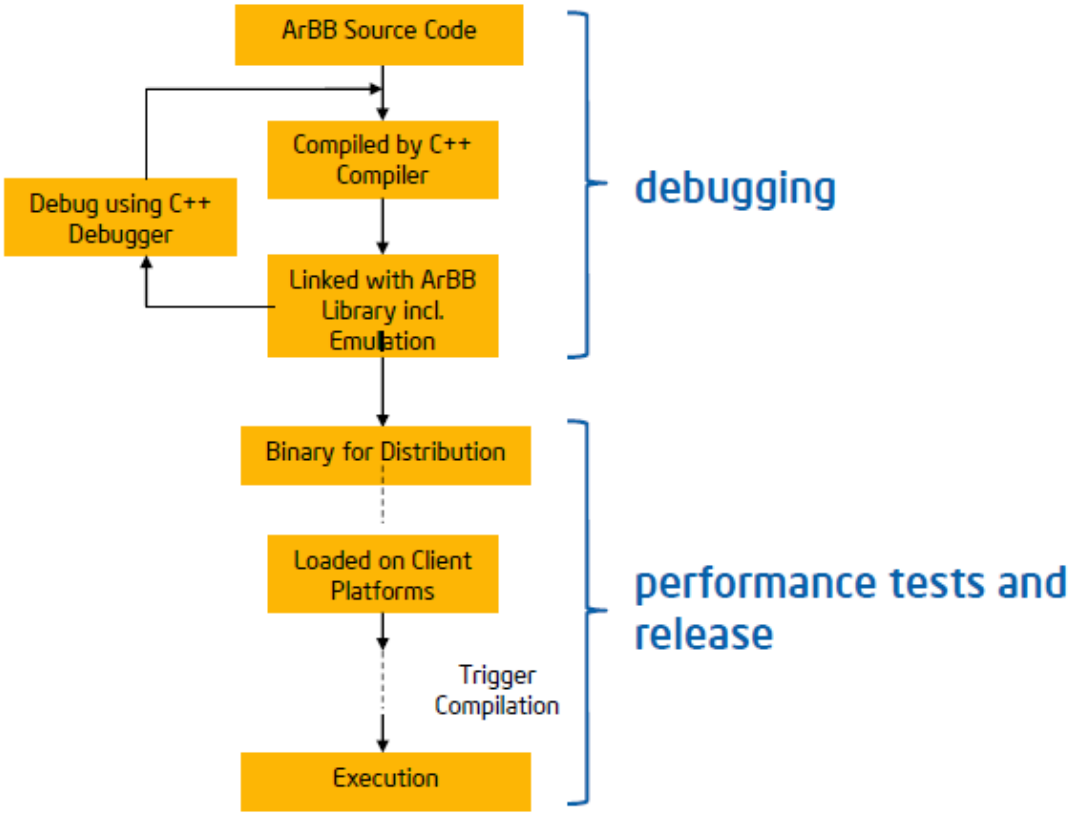


Ilustración 10. Modelo de ejecución en ArBB

A diferencia de otras bibliotecas como OpenMP, la división del trabajo no es tarea del programador por defecto. ArBB siempre opera sobre todos los elementos de la matriz o vector y es tarea de la *JIT Compilation* decidir la división del trabajo durante la compilación dinámica. Sin embargo, el programador tiene a su disposición variables de entorno que permiten modificar el comportamiento de la biblioteca. Estos son algunos de los más importantes[3]:

- **ARBB_OPT_LEVEL:** permite seleccionar el tipo de optimización:
  - O0: código secuencial, útil para tareas de depuración. En este caso solo se ejecuta la primera fase de compilación.
  - O1: permite vectorización pero no aprovecha la arquitectura multiprocesador.
  - O2: permite tanto vectorización como código multi-hilo.
- **ARBB_VERBOSE:** emite mensajes de diagnóstico del compilador durante la ejecución del programa.
- **ARBB_NUM_THREADS:** el programador puede seleccionar el número de hilos.

[3] Además de las variables de entorno, existen funciones que cumplen con la misma tarea. Ejemplos: arbb_set_generate_threaded_code, arbb_set_num_threads y arbb_set_heap_size.

- **ARBB_INIT_HEAP y ARBB_MAX_HEAP:** permiten modificar el tamaño de la pila.

Otro de las cualidades de ArBB es la seguridad, que evita de forma automática los errores de programación comunes que causan condiciones de carrea o bloqueo entre hilos, aspectos muy negativos en el rendimiento de una aplicación, además se encarga del balanceo de carga lo que implica una mejora en la utilización de recursos.

La abstracción que proporciona ArBB se traduce en un cambio de sintaxis más próximo a la notación matemática y al modelo secuencial. Como punto negativo, esta abstracción obliga al programador a utilizar los tipos de datos propios de Array Building Blocks, en la Tabla 3 puede observarse la conversión de algunos de los tipos entre C++ y ArBB.

Tipo	Descripción	Equivalente C++
i8,i16,i32,i64	Número entero con signo de 8/16/32/64 bits.	char, short, int.
f32,f64	Número en coma flotante de 32/64 bits.	float, double
Boolean	Variable booleana	bool
dense<f32> name(N)	Array de una dimensión de tipo f32 de tamaño N	float name [N]
dense<i8,2> name(N,M)	Array bidimensional de tipo char de tamaño N*M	char name[N][M]

Tabla 3. Tipos de variables en ArBB y sus equivalentes

Esta conversión de datos puede dar lugar a confusión pensando que ArBB es un nuevo lenguaje, lejos de todo esto solo se trata de una biblioteca para C++. ArBB incorpora mecanismos para la conversión entre tipos de datos de C++ a ArBB y viceversa, enlaces entre los dos tipos de datos y funciones con las operaciones de paralelización más comunes como *scatter()* o *add_reduce()*. Como punto negativo, ArBB es muy restrictivo en cuanto a la programación, la convivencia entre sintaxis pura de C++ y sintaxis ArBB puede dar lugar a problemas para conseguir el algoritmo deseado.

A continuación se muestra un pequeño ejemplo donde se puede comparar la versión secuencial y la versión con ArBB de un programa que calcula el producto escalar de dos vectores:

Versión C++

```
double producto_escalar (double* vec1, double* vec2){
    double resultado = 0;
    for(int i = 0; i < N; i++){
        resultado += vec1[i]*vec2[i];
    }
    return resultado;
}
int main(int argc, char** argv){
    double vec1[N];
    double vec2 [N];
    double resultado =  0;
    //inicialización de vec1 y vec2;
    resultado = producto_escalar(vec1,vec2);
}
```

Versión ArBB

```
using namespace arbb;
void map_producto_escalar(f64 elemento1, f64 elemento2, f64& elemento 3){
        elemento3 = elemento1 * elemento2;
}
void producto_escalar (dense<f64> vec1, dense<f64 vec2, f64& resultado){
    dense<f64> vec3 = fill(f64(0),N);
    map(map_producto_escalar)(vec1,vec2,vec3);
    resultado = add_reduce(vec3);
}
int main(int argc, char** argv){
    dense<f64> vec1(N);
    dense<f64> vec2(N);
    f64 resultado = 0;
    //inicialización de vec1 y vec2;
    call(producto_escalar)(vec1,vec2,resultado);
}
```

En la versión paralelizada no hay ni bucles que dividan el trabajo, ni sentencias especificando del número de procesadores/tareas, ni el uso de mecanismos de bloqueo típicos en la programación paralela. Ejemplos como este clarifican el nuevo modelo de programación que propone Array Building Blocks bajo el lema *what to do rather than how to do it* [4] .

[4] Qué hacer en lugar de como hacerlo.

3. PLANTEAMIENTO DEL PROBLEMA

En este capítulo se describen un conjunto de requisitos con el fin de definir formalmente las características que deben tener las aplicaciones paralelizadas y los sistemas utilizados para las tareas de desarrollo y evaluación. En este trabajo no se aplica ninguna regulación técnica o legal además de las descritas en los requisitos. No obstante, con el fin de estandarizar el trabajo de implementación se seguirá el estándar internacional C++ ISO/IEC 14882:2011.

3.1.Análisis de requisitos

A modo informativo, las tablas de requisitos tienen la siguiente estructura:

- **Identificador:** campo unívoco para identificar cada requisito.
- **Tipo:** Se distinguirá entre funcional o no funcional.
  - Funcional: especifican el qué tiene que hacer el software.
  - No funcional: imponen restricciones en el diseño.
- **Conflictos:** enumeran los requisitos contradictorios con el requisito en cuestión.
- **Necesidad:** Se distinguirá entre esencial, deseable y opcional.
  - Esencial: de necesidad obligaría en el trabajo a realizar.
  - Deseable: de carácter negociable.
  - Opcional: la no inclusión no supone penalización.
- **Descripción:** breve explicación del contenido del requisito.
- **Justificación:** argumentación del porqué de la inclusión de dicho requisito o la motivación que lo ha propiciado.

Los requisitos obtenidos, tras el análisis, son los siguientes:

Identificador: RNF-01			
Tipo	No funcional	Conflictos	#
Necesidad	☑ Esencial ☐ Deseable ☐ Opcional		
Descripción	En el desarrollo y evaluación de las aplicaciones se utilizará el sistema operativo Linux.		
Justificación	Array Building Blocks sólo es compatible con Linux.		

Tabla 4. Requisito RNF-01

Identificador: RNF-02			
Tipo	No funcional	Conflictos	#
Necesidad	☐ Esencial ☑ Deseable ☐ Opcional		
Descripción	Evaluar las aplicaciones del benchmark PARSEC y seleccionar las que más se ajusten a las características de ArBB.		
Justificación	Explorar las capacidades de Intel Array Building Blocks con las aplicaciones que más se ajusten a las características de este modelo.		

Tabla 5. Requisito RNF-02

Identificador: RNF-03			
Tipo	No funcional	Conflictos	#
Necesidad	☐ Esencial ☑ Deseable ☐ Opcional		
Descripción	Las aplicaciones seleccionadas del PARSEC deberán ser compatibles con el sistema operativo Linux y el compilador gcc-4.4.		
Justificación	Unificar y estandarizar el entorno de desarrollo y evaluación.		

Tabla 6. Requisito RNF-03

Identificador: RNF-04			
Tipo	No funcional	Conflictos	#
Necesidad	☑ Esencial ☐ Deseable ☐ Opcional		
Descripción	Las aplicaciones paralelizadas con ArBB deberán permitir la utilización de distinto número de cores.		
Justificación	La aplicación debe ser evaluada con distintos cores.		

Tabla 7. Requisito RNF-04

Identificador: RNF-05			
Tipo	No funcional	Conflictos	#
Necesidad	☑ Esencial ☐ Deseable ☐ Opcional		
Descripción	En caso de utilizar el modelo Threading Building Blocks junto Array Building Blocks en la implementación de las aplicaciones será necesario mantener un compromiso máximo de 70%-30% entre Intel TBB e Intel ArBB.		
Justificación	El Trabajo Fin de Grado está enfocado a la utilización principal de Array Building Blocks.		

Tabla 8. Requisito RNF-05

Identificador: RNF-06			
Tipo	No funcional	Conflictos	#
Necesidad	☐ Esencial ☑ Deseable ☐ Opcional		
Descripción	La evaluación del rendimiento deberá ser probado en distintas arquitecturas.		
Justificación	Conocer el comportamiento de ArBB en distintos entornos.		

Tabla 9. Requisito RNF-06

Identificador: RNF-07			
Tipo	No funcional	Conflictos	#
Necesidad	☐ Esencial ☑ Deseable ☐ Opcional		
Descripción	La aceleración obtenida en las aplicaciones paralelizadas con ArBB deberá ser igual o superior a 2, respecto a la versión secuencial, para las entradas *simlarge* y *native* (entradas del benchmark PARSEC) en todas las arquitecturas probadas.		
Justificación	Deben conseguirse resultados positivos en la paralelización a realizar.		

Tabla 10. Requisito RNF-07

Identificador: RNF-08			
Tipo	No funcional	Conflictos	#
Necesidad	☑ Esencial ☐ Deseable ☐ Opcional		
Descripción	Evaluar el rendimiento de las aplicaciones paralelizadas con el resto de modelos de programación paralela.		
Justificación	Conocer el rendimiento de ArBB con el resto de modelos para conocer su rendimiento.		

Tabla 11. Requisito RNF-08

Identificador: RF-09			
Tipo	Funcional	Conflictos	#
Necesidad	☑ Esencial ☐ Deseable ☐ Opcional		
Descripción	Los argumentos pasados en la versión secuencial deberán ser los mismos para la versión con ArBB.		
Justificación	Mantener la estructura propuesta en la versión secuencial.		

Tabla 12. Requisito RF-09

Identificador: RF-10			
Tipo	Funcional	Conflictos	#
Necesidad	☐ Esencial ☑ Deseable ☐ Opcional		
Descripción	Verificar que los resultados obtenidos en las aplicaciones paralelizadas no introducen errores.		
Justificación	Los resultados deben ser iguales respecto la versión secuencial existente en el benchmark PARSEC.		

Tabla 13. Requisito RF-10

4. DISEÑO DE LA SOLUCIÓN TÉCNICA

Dentro de este capítulo se describen las optimizaciones realizadas en ambas aplicaciones, así como el proceso seguido para obtener la solución final implementada. Ambos diseños se han realizado teniendo en cuenta las características técnicas de Array Building Blocks mencionadas en el punto 2.3.4. Como punto inicial se tomarán las versiones secuenciales de Blackscholes y Fluidanimate que vienen incluidas en la versión 2.1 de la suite del benchmark PARSEC.

4.1.Blackscholes

El algoritmo de Blackscholes es muy favorable para la paralelización de datos que se pretende. La compartición y el intercambio de información, entre las distintas opciones financieras, son nulos, por lo que si se contase con un número de procesadores igual al número de opciones podría ejecutarse todo el proceso de forma paralela, es decir, no hay limitaciones de paralelismo. La resolución del modelo de Black-Scholes es una operación compleja debido a la resolución de una ecuación diferencial lo que se traduce en la utilización de números en coma flotante y la posibilidad de aplicar las instrucciones vectoriales que incluye el modelo Array Building Blocks.

El código con la versión secuencial de Blackscholes contiene, además de esta, la versión para OpenMP y Pthreads. Por lo tanto, el primer paso será eliminar el código paralelizado con estas bibliotecas y obtener un código secuencial limpio. Blackscholes puede ejecutarse tanto en sistemas operativos Windows como Linux, el sistema operativo utilizado para la implementación y evaluación es Linux por lo que todo el código implementado para Windows debe ser eliminado también. Una vez realizado esto ya se dispone de la versión secuencial que se tomará como punto de partida para el análisis de la aplicación.

Blackscholes utiliza una estructura denominada *OptionData* que conserva la información relevante de cada opción financiera. Esta estructura está compuesta por un total de 7 *floats* y 1 *char*, por lo que la información que computa la aplicación vendrá determinada por un array de este tipo de estructura con una longitud determinada por el fichero de entrada.

Para realizar un correcto diseño de las optimizaciones que se pueden realizar a la aplicación, es necesario identificar las distintas partes del código. Esta es la división obtenida tras el análisis:

- **Fase inicial y lectura de la información:** verifica que el número de hilos es uno y comprueba que el fichero de entrada puede abrirse correctamente. Posteriormente, y haciendo uso de un array de *OptionData* lee y guarda la

información del fichero de texto. Acto seguido desgrana esta información en arrays separados.

- **Fase de cómputo:** Mediante un bucle, el programa realiza el cómputo de cada opción de manera individual a través de la función *BlkSchlsEqEuroNoDiv*(), donde 2 de los 7 *floats* no son utilizados en esta optimización. Esta función es la que se encarga de realizar todos los cálculos necesarios para obtener el precio final. La función *BlkSchlsEqEuroNoDiv()* utiliza una función auxiliar denominada *CNDF()* que se encarga de calcular la distribución acumulativa de la distribución normal y permite la resolución de la ecuación de Black-Scholes. La fase de cómputo completa es repetida en función de la variable NUM_RUNS. NUM_RUNS expresa el número de veces que se realizará la misma operación con el fin de reducir la influencia de la parte secuencial de la aplicación, que se corresponde con la fase de lectura y escritura de resultados.
- **Fase final y escritura de los resultados:** Tras guardar en un array de *float* los resultados, la aplicación escribe la información en un fichero de texto y libera la memoria utilizada en los arrays. Una vez realizado este paso la ejecución del programa finaliza.

De las tres fases mencionadas solo la fase de cómputo puede ser paralelizada, y es en esta donde se han invertido todos los esfuerzos. Tanto la lectura como la escritura de ficheros es un proceso secuencial que no es objetivo de este trabajo.

El primer paso, tras analizar el programa y detectar las zonas que pueden ser explotadas, es crear los tipos de datos ArBB necesarios para realizar el cómputo. La entrada y salida de la aplicación viene dada por un fichero de texto, por lo que la forma más eficiente para realizar la ejecución de las tres fases es conservar las variables antiguas para realizar las operaciones de entrada y salida y crear variables de tipo ArBB para el cómputo. Gracias a la función *bind()* es posible enlazar tipos de datos de C++ con tipos de datos ArBB, esto permite tener la misma información en dos tipos de variables distintas que se actualizan de forma dinámica, gracias a esto no es necesaria una copia masiva entre variables. Como se mencionó anteriormente, la función *BlkSchlsEqEuroNoDiv()* solo utiliza cinco de las siete *floats* de cada opción, para las dos variables restantes no es necesario crear variables de tipo ArBB. Esta función devuelve como resultado un número en coma flotante, por lo que también es necesario crear una variable ArBB para su salida.

Tras finalizar el proceso de creación de variables solo queda diseñar la función *BlkSchlsEqEuroNoDiv()* y su auxiliar *CNDF()* para ArBB. El *char* que posee cada opción se utiliza para determinar el tipo de operación a realizar (compra o venta), además el algoritmo contempla dos acciones distintas en función del signo de los resultados obtenidos, lo que implica que el tratamiento de cada opción sea distinto en función de del valor de sus variables por lo que no se conseguirá un paralelismo de datos puro. La solución a este inconveniente se resuelve gracias a la función *map()* y a la sentencia *_if* de ArBB. Esto permite operar de forma individual cada opción teniendo en cuenta las restricciones comentadas.

El proceso es el siguiente: la función *BlkSchlsEqEuroNoDiv_call()* de ArBB que posee los arrays de datos con todas las opciones llama a la función *BlkSchlsEqEuroNoDiv_map()* que será la encargada de operar las opciones de forma independiente. Uno de los problemas de ArBB es que dentro de la función *map()* no se puede llamar a ninguna otra función. En el algoritmo original la función *BlkSchlsEqEuroNoDiv()* llamaba a la función *CNDF()* para cada opción, en este caso no es posible. Siguiendo uno de los consejos de los ingenieros de Intel, *Make ArBB functions as large as posible*[5], se fusiona la función *CNDF()* con *BlkSchlsEqEuroNoDiv_map()* quedando una única y gran función. Gracias a se solventa el problema de la llamada a otra función y se crean más oportunidades para que la máquina virtual de ArBB genere una mejor optimización.

El resto del proceso es bien sencillo, ya que *BlkSchlsEqEuroNoDiv_map()* no es más que una traducción del algoritmo del Blackscholes secuencial, incluyendo la función *CNDF()*, utilizando la sintaxis de ArBB. Como se mencionó anteriormente, la aplicación ejecuta el proceso de cómputo un determinado número de veces fijado en la variable NUM_RUNS[6]. ArBB captura las funciones la primera vez que son llamadas, esto supone una gran inversión de tiempo la primera iteración pero permite que las sucesivas sean mucho más rápidas, cuanto mayor sea NUM_RUNS la inversión inicial en la captura de la función será más rentabilizada.

Tras finalizar la implementación y comparar los resultados obtenidos, entre la versión secuencial y la implementada, hay una pequeña diferencia entre ambos. El error obtenido en la función *BlkSchlsEqEuroNoDiv_map()* se debe a la discrepancia de resultados entre la función *arbb::exp()* que incorpora la biblioteca de Intel Array Building Blocks y la función *exp()* de la biblioteca *math.h* de C. La diferencia obtenida entre ambas versiones es despreciable como se puede apreciar en la Tabla 14.

	Fichero in_64K
Número de opciones	65.536
Suma de errores	0,290063049
Error máximo	4,19617E-05
Error mínimo	0
Promedio	4,42601E-06
% de diferencia entre ArBB y secuencial	0,000055849 %

Tabla 14. Errores introducidos en Blackscholes con ArBB

Una vez comprobado que los resultados obtenidos son correctos, solo queda evaluar el impacto de la aplicación implementada con ArBB en una arquitectura multicore y comprobar si realmente la versión implementada hace uso de varios cores. La herramienta Intel® VTune™ Amplifier XE[17] de Intel permite analizar el comportamiento de aplicaciones paralelizadas mediante los modelos TBB y ArBB. El

[5] Crear funciones ArBB tan grandes como sea posible.
[6] Por defecto 100.

análisis se ha realizado con una entrada que simula una situación real con 500 iteraciones para evitar una gran interferencia de las fases secuenciales de lectura y escritura. El computador utilizado tiene un total de 8 núcleos (punto 5.1.1). Los resultados, tras el análisis de Vtune, puede observarse en la Ilustración 11. En esta ilustración aparecen tres gráficos, el primero de ellos muestra el tiempo de cómputo de cada core, el segundo gráfico muestra el cómputo total en cada momento de la ejecución y el último gráfico describe la utilización de varios cores de forma simultánea.

El análisis muestra resultados satisfactorios.. Dentro de la fase de cómputo, cada máximo local se asocia a una iteración de la aplicación y la media del uso de la CPU está próxima al 800%, por lo que se utilizan todos los recursos disponibles. Otro dato interesante es la calificación que aporta VTune al código implementado en cuanto al uso de la CPU. En esta valoración un 84,4% lo considera excelente, el 14,4% es calificado como pobre debido, en gran medida, a la parte secuencial y un 1,1% como aceptable ya que algunas de las iteraciones no aprovechan todos los recursos.

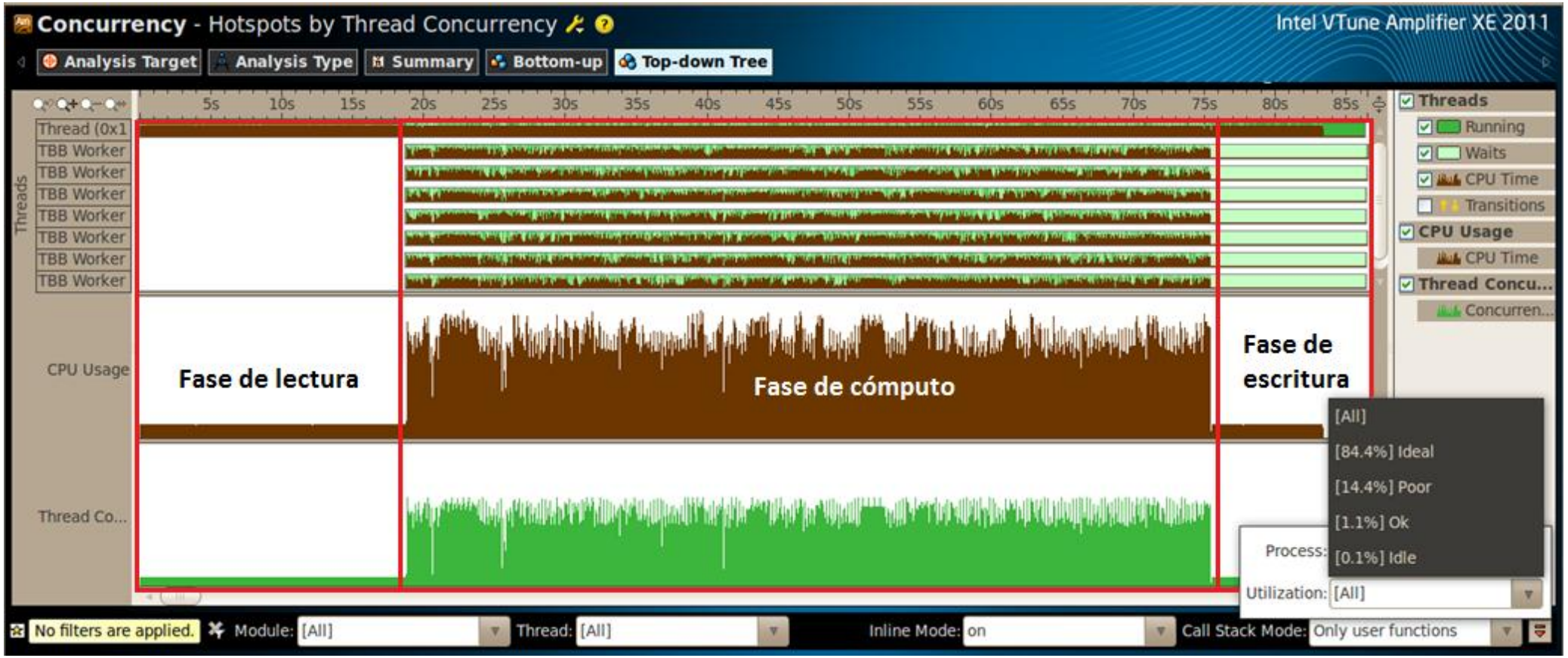


Ilustración 11. Resultados de la herramienta VTune para Blackscholes

4.2.Fluidanimate

La paralelización de esta aplicación mediante el modelo de Intel Array Building Blocks es una tarea complicada debido a la dificultad de adaptar las necesidades de la aplicación a los requisitos del modelo. Tras el estudio del código secuencial, el primer paso es dividir la aplicación e identificar las zonas de código que se podrían ser explotadas. La segmentación del código se ha realizado basándose en las distintas funciones y teniendo en cuenta el tiempo de cómputo de cada una de estas, mediante un *profiler*. Un *profiler* es una herramienta que permite analizar el comportamiento de la aplicación mostrando el tiempo de cómputo invertido en cada función, en este análisis se ha utilizado la herramienta Gprof[18] de Linux para códigos escritos en C y C++. El fichero utilizado para el análisis ha sido el correspondiente a la entrada *simlarge* del PARSEC, con un total de 300.000 partículas y una única iteración. Estas son las fases identificadas:

- **Fase inicial y lectura de la información:** Mediante la función *InitSim()* el programa lee de un fichero binario, pasado por argumento, la información de cada partícula. Esta operación solo se realiza una vez por cada ejecución de Fluidanimate. *InitSim()* ocupa en torno a un 8% del tiempo de ejecución total de la aplicación. Esta fase no será paralelizada puesto que la lectura de la información debe realizarse de forma síncrona para poder situar cada partícula en su lugar correspondiente dentro de la matriz tridimensional. Además, la interferencia de esta fase secuencial no es muy elevada por lo que no supondrá ningún problema para obtener un buen rendimiento tras la paralelización.
- **Fase de cómputo:** Mediante la función *AdvanceFrame()* se ejecutan las siguientes funciones:
    - RebuildGrid: Esta función reposiciona las partículas dentro de la cuadricula tridimensional tras el procesado. Para la recolocación aplica ciertos filtros que marcan los límites de los márgenes de la matriz y de cada celda. El tiempo de cómputo de esta función es de un 6% respecto al total. Esta operación si se repite N veces y el orden de ejecución no importa, la paralelización de esta función si puede ser llevada a cabo.
    - ComputeForces: Dentro de esta compleja función se lleva a cabo la interacción entre partículas determinando su nueva densidad y aceleración. El tiempo de cómputo está en torno a un 68% del tiempo total. Obviamente, esta función será clave para conseguir una buena optimización de la aplicación.
    - ProcessCollisions: Esta función modifica la trayectoria de alguna partícula que está excediéndose de los límites de la matriz tridimensional. La función no alcanza el 5% del tiempo de cómputo total, teniendo en cuenta la ejecución iterativa de esta función podría llevarse a cabo su paralelización sin problemas aparentes y con buenos resultados.

- AdvanceParticles: La última función de *AdvanceFrame()* se encarga de actualizar los parámetros de cada partícula en función del tiempo transcurrido. Su tiempo de cómputo es de un 7% respecto al total, al igual que el resto de funciones de esta fase *AdvanceParticles()* debería ser paralelizada para reducir el tiempo de cómputo.

La carga de trabajo en esta fase es elevada, con una única iteración su tiempo de cómputo asciende a más del 85%. El alto tiempo de cómputo invertido y la iteración de esta fase supone una paralelización prácticamente total, lo que evita problemas acarreados por las secuenciales.

- **Fase de escritura de resultados:** Esta última fase se encarga de la escritura de los resultados procesados en el punto anterior. Aunque esta fase es opcional, ha sido incluida en el análisis. *SaveFile*(), que es la función que se encarga de la escritura de los resultados, ocupa alrededor de un 6% del tiempo de cómputo total. La justificación del porqué no se tiene en cuenta esta función para ser paralelizada es similar a la de la fase de lectura, la escritura debe ser síncrona y el tiempo de cómputo no es muy elevado como para tenerlo en cuenta.

4.2.1. Optimización inicial y puesta a punto

El código de Fluidanimate original, incluido en el benchmark PARSEC, sólo puede ser ejecutado de forma secuencial, este será el código de partida para comenzar la optimización mediante ArBB. No obstante, hay algunos cambios que se pueden realizar con el fin de evitar redundancias o cálculos innecesarios.

La versión incluida en el PARSEC tiene fines académicos, por lo que permite ajustar una gran cantidad de parámetros. En esta ocasión, se limitará la diversidad de la aplicación reduciendo al código estrictamente necesario. Muestra de esta redundancia es la utilización de variables y funciones que posteriormente no son usadas o no tienen utilidad en el propio código, como por ejemplo las funciones *bswap_float()* o *isLittleEndian()*.

Otra de las modificaciones realizadas es la eliminación de cálculos innecesarios. Un ejemplo de esto es el cálculo de la variable *tc*, que es igual a la variable *hSq* al cubo, y que se calcula cada vez que se llama a la función *ComputeForces()*. Este cálculo es inútil, ya que *hSq* no varía a lo largo de la ejecución del programa, como solución podría calcularse el valor de *tc* únicamente al inicio de la ejecución del programa.

Por último, se ha modificado la estructura original del código para que sea más legible y más adaptable a la paralelización. El principal cambio realizado es el desglose de la función *ComputeForces()* en cuatro pequeñas funciones: *InitAcelerationAndDensity()*, *ComputeDensities()*, *ModifyDensity()* y *ComputeForces()*. La puesta a punto realizada permite disponer de un código adaptado y legible para la tarea de paralelización.

4.2.2. Versión 1.0

El diseño de la solución de Fluidanimate con ArBB inicialmente era similar al planteado en Blackscholes. Tal y cómo se enuncia en este nuevo modelo de Intel y cómo se realizó con Blackscholes, la paralelización se basa en la traducción del código escrito en C++ al código ArBB teniendo en cuenta los factores que pueden afectar en el rendimiento de la aplicación.

Para la versión 1.0, primeramente se realizó el proceso de adaptación a ArBB. Esto supone la modificación de todas las variables que vayan a ser utilizadas en el algoritmo paralelizado (Tabla 3) y la adaptación de las funciones correspondientes. Fluidanimate utiliza una estructura denominada *Cell* compuesta por objetos *Vec3* que simulan un vector tridimensional. ArBB solo tiene equivalentes a tipos simples como *float* o *int* lo cual implica la necesidad de destruir esta estructura junto con la clase *Vec3*, y reducirlo todo a tipos de variables simples. La dificultad de esta tarea es mínima pero sus consecuencias no lo son tanto. Con la medida adoptada se pasa de utilizar dos arrays de estructuras *Cell* a veinte arrays de *floats* y diez *arbb::denses* de tipo *f32*, lo cual supone una mayor complejidad de las funciones y la necesidad de aumentar el tamaño del programa.

Una vez superado el problema de la equivalencia entre variables, se plantea un nuevo reto, el acceso a las partículas o lo que es igual la dimensión de los arrays. Es muy importante tener en cuenta cómo se va a recorrer la matriz de celdas, ya que está fuertemente relacionado con los fallos de caché. Un recorrido en el orden equivocado podría dar lugar a resultados nefastos en cuanto al tiempo de cómputo. Tras las pruebas realizadas se ha comprobado cómo el *arbb::dense* de una dimensión introduce un mejor rendimiento que el *arbb::dense* de dos o tres dimensiones por lo que se utilizarán elementos unidimensionales.

Tras definir las nuevas variables y siguiendo el proceso utilizado con Blackscholes solo queda llevar al cabo la traducción de funciones. Con el fin dividir el trabajo y comprobar el rendimiento de forma paulatina se paralelizaron las funciones *RebuildGrid(), InitAcelerationAndDensity()* y *ComputeDensities()*. Tras la implementación y evaluación de los resultados aparecieron nuevas dificultades.

La matriz tridimensional, donde se encuentran las partículas en Fluidanimate, contiene celdas con menos de dieciséis partículas, llegando incluso a existir celdas vacías sin partículas. La versión secuencial utiliza un array auxiliar que contabiliza el número de partículas de cada celda para así operar únicamente en las posiciones de partículas existentes. ArBB otorga una alta abstracción al programador pero de forma interna está diseñado para operar la totalidad de la matriz, esta restricción insalvable supone un desperdicio de los recursos y por tanto una peor mejora del rendimiento. Tras detectar este problema, es interesante identificar cual es el nivel real de ocupación en la matriz tridimensional para los distintos ficheros de entrada existentes. Los ficheros de entrada se han dividido en dos grupos, ficheros de menor tamaño (*test, simdev y simsmall*) y

ficheros de gran tamaño (*simmedium, simlarge y native*). El nivel de ocupación de partículas se resume en la Tabla 15.

	Ficheros pequeños	Ficheros grandes
Celdas con partículas	15%	45%
Celdas completas (16 partículas)	0,5%	0%
% de ocupación total	**6,5%**	**15%**

Tabla 15. Nivel de ocupación en los ficheros originales de Fluidanimate

Los porcentajes de ocupación son muy pequeños lo que supone un gran desperdicio del tiempo de cómputo. La inclusión de un array auxiliar, al igual que en la versión secuencial, junto con numerosas sentencias *_if* permite obtener los resultados deseados pero no evita que ArBB tenga en cuenta toda la matriz y se reduzca el tiempo invertido en cada tarea. Una solución planteada al problema sería la incorporación de matrices dispersas existentes en ArBB y denominadas *arbb::nested*. Con estos contenedores irregulares podría solventarse el problema planteado. El uso de *arbb::nested* fue desestimado debido a que las partículas dentro de la matriz van mutando por cada iteración, lo que obligaría a declarar una definir un nuevo contenedor por cada iteración, además la utilización de matrices dispersas dificulta la búsqueda de partículas vecinas en la función *ComputeDensities()*.

En la función *ComputeDensities()* se realiza una de las interacciones entre partículas de la aplicación. En ella, es necesario buscar las partículas vecinas a una dada. En una función ArBB, el programador no puede seleccionar un elemento del array ya que el algoritmo se aplica de forma homogénea a todos los elementos, tal y cómo se muestra en el ejemplo mostrado en el punto 2.3.4. En la búsqueda de vecinos es necesario saber la posición de cada partícula para así determinar si el elemento está en el límite de la matriz y para conocer las posiciones de sus vecinos, esta información se puede obtener gracias a la función *position()* de ArBB, el acceso a los vecinos se puede realizar a través de la función *neighbour()*. En la API de ArBB se menciona que el uso de la función *position()* puede suponer una sobrecarga en los tiempos de cómputo, tras probar algunos ejemplos se ha comprobado cómo esta función imposibilita la optimización del código, ya que introduce pérdidas muy superiores a las permitidas[7].

El último de los problemas presentados se debe a la utilización de los bucles *_for* proporcionados en la biblioteca de ArBB. Con el fin de traducir el código íntegro del original al escrito en ArBB, se pasó de utilizar bucles *for* de C++ a utilizar bucles *_for* de ArBB. Estos bucles son necesarios para realizar la búsqueda de vecinos y poder recorrer la matriz tridimensional, además los bucles *for* de C++ no permiten utilizar variables ArBB, por lo que su utilización es obligada. El problema es que este tipo de bucles están diseñados para expresar código secuencial[19], es decir, todo el código englobado dentro de un bucle *_for* será ejecutado de forma secuencial. El objetivo es conseguir obtener la mayor paralelización posible y la utilización de este tipo de bucles no aporta resultados efectivos.

[7] En los ejemplos realizados la aceleración pasa de ser 8 sin la función position a ser de 0,2 con dicha función.

Como resumen final, esta primera versión no arroja resultados válidos para los objetivos propuesto puesto que su tiempo de cómputo es incluso mayor que el de la versión secuencial. Sin embargo, esta primera versión puede ser tomada como punto de referencia para posteriores versiones y gracias a ella se ha conseguido un primer acercamiento a Fluidanimate con Intel Array Building Blocks.

4.2.3. Versión 2.0

Una vez comprobada en la versión 1.0 las restricciones establecidas por ArBB y las necesidades de Fluidanimate, es momento de diseñar una solución válida para los problemas planteados.

Array Building Blocks es un modelo orientado a optimizar operaciones vectoriales y en él no tienen en cuenta un vector incompleto como es el caso de Fluidanimate. Prueba de ello es la función *map()*, encargada de la paralelización del código, que se encarga de aplicar la misma operación a todos los elementos de la matriz. Tras desestimar el uso de *arbb::nested* por la imposibilidad de su utilización, será necesario operar la totalidad de la matriz a pesar de desperdiciar recursos. Durante la fase de evaluación se determinará cuál es el nivel de ocupación de partículas necesario para que ArBB pueda conseguir un buen rendimiento. Así pues, las funciones *RebuildGrid()*, *InitAcelerationAndDensity()*, *ModifyDensity()*, *ProcessCollisions()* y *AdvanceParticles()* podrán ser optimizadas con ArBB teniendo en cuenta que la matriz tridimensional será operada a la totalidad. En estas funciones, las operaciones que se realizan sobre una partícula no tienen consecuencias sobre el resto, así pues, tras computar todas las posiciones de la matriz bastará con desechar las posiciones inhabitadas.

Para no entrar en detalles del propio código, se mostrará un ejemplo de la modificación que sufren las funciones tras ser optimizadas. En dicho ejemplo se ilustra la función *AdvanceParticles()* tanto en la versión secuencial como la nueva versión con ArBB.

Versión secuencial en C++

```
void AdvanceParticles(){
  for(int i = 0; i < numCells; ++i){
      Cell &cell = cells[i];
      int np = cnumPars[i];
      for(int j = 0; j < np; ++j){

          Vec3 v_half = cell.hv[j] + cell.a[j]*timeStep;

          cell.p[j] += v_half * timeStep;

          cell.v[j] = cell.hv[j] + v_half;

          cell.v[j] *= 0.5f;

         cell.hv[j] = v_half;
          }
     }
}

void AdvanceFrame(){
  …//resto de funciones
  AdvanceParticles();
}
```

Versión 2.0 en ArBB

```
void AdvanceParticles_map(f32& px, f32& py,
f32& pz, f32& ax, f32& ay, f32& az, f32& hvx,
f32& hvy, f32& hvz, f32& vx, f32& vy, f32& vz){

        f32 v_halfX = hvx + ax*timeStep;
        f32 v_halfY = hvy + ay*timeStep;
        f32 v_halfZ = hvz + az*timeStep;

        px+=v_halfX*timeStep;
        py+=v_halfY*timeStep;
        pz+=v_halfZ*timeStep;

        vx = (hvx+v_halfX)*0.5;
        vy = (hvy+v_halfY)*0.5;
        vz = (hvz+v_halfZ)*0.5;

        hvx = v_halfX;
        hvy = v_halfY;
        hvz = v_halfZ;
}

void AdvanceParticles_call(dense<f32>& px,
dense<f32>& py, dense<f32>& pz,
dense<f32>& ax, dense<f32>& ay,
dense<f32>& az, dense<f32>& hvx,
dense<f32>& hvy, dense<f32>& hvz,
dense<f32>& vx, dense<f32>& vy,
dense<f32>& vz){

        arbb::map(AdvanceParticles_map)(px,py,p
        z,ax,ay,az,hvx,hvy,hvz,vx,vy,vz);
}

void AdvanceFrame(){
        …//resto de funciones
        call(AdvanceParticles_call)(densePx,
        densePy, densePz, denseAx, denseAy,
        denseAz, denseHVx, denseHVy,
        denseHVz, denseVx, denseVy, denseVz);
}
```

Cómo se mencionó en la versión 1.0, la complejidad del algoritmo no aumenta al utilizar ArBB, aunque sí aumenta el número de variables y por tanto la longitud del código. En la versión con ArBB es necesario utilizar una función intermedia denominada *call()* a la que se le pasa el array entero, dentro de esta función podría

ejecutarse código ArBB pero en esta solución. La función *call()* se encarga de llamar a la función *map()* que operará cada posición del array de forma independiente, la función *call()* pasa por parámetro a la función *map()* el array entero, pero esta solo recibe un *f32* (*float* en ArBB). Este será el proceso a seguir para el resto de funciones mencionadas anteriormente.

Tas la optimización de estas funciones solo queda por paralelizar las funciones *ComputeDensities()* y *ComputeForces()*. Estas funciones merecen especial atención puesto que en ellas se invierte gran parte del tiempo de cómputo de la aplicación, en ellas se desarrolla el algoritmo de interacción de partículas similar a la operación *Stencil* descrita en el punto 2.1.2.1. El pseudocódigo de estas funciones es el siguiente:

```
celdasVecinas[27]
indiceCelda = 0
numeroCeldasVecinas = 0
k = 0
Mientras k < dimensiónEjeZ Hacer
    j = 0
    Mientras j < dimensiónEjeY Hacer
        i = 0
        Mientras i < dimensiónEjeX Hacer
            Numero de partículas = numParticulas[indiceCelda]
            Si Numero de partículas = 0 Entonces
                Continuar
            Fin Si
            numeroCeldasVecinas = ObtenerCeldasVecinas(k, j, i, celdasVecinas)
            celda = celdas[indiceCelda]
            particula = 0
            Mientras particula < Numero de partículas Hacer
                numeroVecino = 0
                Mientras numeroVecino < numeroCeldasVecinas Hacer
                    CeldaVecina = celdasVecinas[numeroVecino]
                    Numero de partículasVecinas = numParticulas[indiceCelda]
                    particulaVecina = 0
                    Mientras particulaVecina < Numero de partículasVecinas Hacer
                      //interacción de partículas entre particula y particulaVecina
                      particulaVecina = particulaVecina + 1
                    Fin Mientras
                  numeroVecino = numeroVecino + 1
                Fin Mientras
                particula = particula + 1
            Fin Mientras
            i = i +1
            indiceCelda = indiceCelda + 1
        Fin Mientras
        j = j +1
    Fin Mientras
    z = z +1
Fin Mientras
```

Se recorre la matriz ZxYxX, eje por eje, para operar cada celda.

Si la celda está vacía se deshecha, en caso contrario se buscan todos sus vecinos, similar a un *Stencil* de 27 puntos.

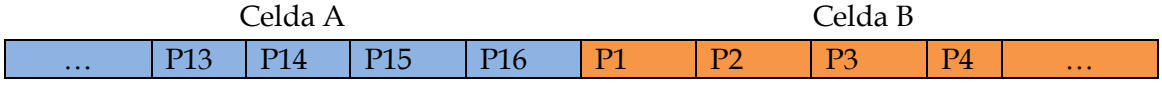
Para cada partícula de una celda se accede a las celdas vecinas. Para cada celda vecina se obtienen sus partículas que interactúan con la partícula original.

La única diferencia entre ambas funciones es la magnitud a computar, siendo la densidad en un caso y la aceleración en otro. Como se vio en la versión 1.0 implementada, el código secuencial no puede ser traducido directamente. Los principales problemas que presentan estas funciones son dos. El primero de ellos es que a diferencia del resto de funciones, las partículas comparten información entre ellas lo que implica que no sea posible operar todos los elementos de la matriz y más tarde desechar las partículas inexistentes. Como posible solución a este problema podría añadirse un array auxiliar compuesto de booleanos y que especificase si cierta partícula existe o no y actuar en consecuencia. La ejecución de la solución planteada obligaría a introducir muchas sentencias *_if* que provocan pérdidas de rendimiento.

El otro inconveniente que dificulta la paralelización es el desconocimiento de la posición de cada elemento computado. Sin saber esta información, no podrían distinguirse las celdas situadas en los límites de la matriz y el acceso a los vecinos sería erróneo, ya que estas celdas tienes menos de los 27 vecinos que poseen el resto de celdas. El objetivo que pretende ArBB es que los accesos a los vecinos sean a partir de un desplazamiento dado respecto la posición actual, en este caso los vecinos están predeterminados y desde la posición actual solo habría que ir desplazándose en los tres ejes x,y,z desde -1 hasta 1 para acceder a los 27 vecinos. El problema es que para celdas situadas en los límites algunas de estas combinaciones son inválidas.

Para superar el problema planteado se presentó una alternativa algo ineficaz desde el punto de vista del rendimiento. Para solucionar la búsqueda de vecinos para las celdas que se encuentran en los límites de la matriz el procedimiento a seguir es simple. Puesto que no es necesario seguir ningún orden al recorrer la matriz (*Stencil* tipo Jacobi) se podrían recalcular las densidades y aceleraciones de forma secuencial de este tipo de celdas especiales tras el cálculo paralelizado previo. El punto negativo a esta solución es que se estaría desperdiciando tiempo de cómputo replicando el cálculo de parte de las partículas de la matriz. Como punto positivo recalcar que las celdas situadas en los límites solo son una minoría del total, lo que no supondría una gran pérdida.

A raíz de esto, si no se puede distinguir cual es la posición de la partícula dentro de la celda el coste de acceso a las partículas vecinas no es uniforme. El acceso a la información de otro elemento en Array Building Blocks se realiza a través de la función *arbb::neighbour()* que permite leer los valores de otros elemento pasándole por argumento un determinado desplazamiento a partir de la posición actual, el problema es que este desplazamiento es variable. Un ejemplo de esto es el siguiente:



El coste para acceder desde la partícula 13, de la celda A, a la partícula 1, de la celda B, es de 4 movimientos por lo que la función *neighbour* sería *neighbour(celdas, 4)* . Pero el

coste para acceder a esta misma partícula desde la partícula 14, de la celda A, es de 3 movimientos, en este caso la función *neighbour* sería *neighbour(celdas, 3)*. La diferencia en el desplazamiento implica accesos no uniformes y esto a su vez obliga a la utilización de la función *postion()* que introduce sobrecargas prohibitivas.

Queda claro que la utilización de la utilización de la variable *position()* es inviable y que el objetivo es conseguir un acceso uniforme para pasarle por argumento a la variable *neighbour()*. Para conseguir que el coste de acceso a las partículas vecinas sea siempre el mismo el problema a resolver debe tener la estructura plena de un *Stencil* de 27 puntos. El algoritmo incluido en Fluidanimate tenía cierta semejanza, ya que la búsqueda de los 27 vecinos si es predeterminada, la única diferencia que aleja el problema planteado de un *Stencil* se presenta a la hora de realizar el cálculo de fuerzas entre partículas, donde el acceso no es predeterminado. Así pues, el objetivo es modificar el cálculo de fuerzas para que de alguna manera el coste sea siempre el mismo. El algoritmo permite variar el orden del cálculo de fuerzas sin modificar el resultado, esta conmutatividad amplia el rango de opciones. Analizando que accesos a partículas son siempre iguales hay un patrón común que se repite entre partículas con la misma posición dentro de la celda. Véase el siguiente ejemplo:

El coste para acceder desde la partícula 5, de la celda A, a su homónima vecina de la celda B es de 16 movimientos. Este patrón se repite para acceder desde la partícula 6, de la celda A, a la partícula 6 de la celda B, así como para el resto de partículas.

Si en lugar de tener un conjunto de celdas se dividiese la matriz tridimensional en 16 pequeñas matrices compuestas de partículas con la misma posición el problema planteado cambiaría, y es que en este caso, tanto la búsqueda de partículas vecinas como el cálculo de sus fuerzas tendría un coste uniforme. Ahora queda por resolver el cómputo de fuerzas para partículas de distinta posición, en este caso los costes no pueden ser uniformes así que la única medida posible es realizar estos cálculos de forma secuencial o recurrir a otros modelos de programación paralela. Intel TBB es un modelo basado en tareas (punto 2.3.3) y que sustenta la base de Intel ArBB, ya que este último utiliza los mismos hilos de ejecución basados en Intel TBB. Por estos motivos, TBB podría ser una buena alternativa para solucionar el cálculo de las densidades y aceleraciones de partículas con diferentes posiciones.

Desde el punto de vista lógico, la solución planteada es viable, pero hay algunos aspectos que es necesario tener en cuenta para la implementación final. Dividir la matriz general de celdas en 16 sub-matrices de partículas provoca una mayor complejidad del código. Una alternativa es utilizar estas sub-matrices durante toda la aplicación, pero implicaría un caos en cuanto a la tarea de programación. Por ejemplo: el cálculo de fuerzas habría que hacerlo a mano, puesto que se cuentan con 16 arrays

unidimensionales distintos sería necesario realizar 240[8] iteraciones a cargo del programador (partículas de la posición uno con partículas de la posición dos, partículas de la posición uno con partículas de la posición tres, etc.). Además obligaría a modificar el algoritmo de todas las funciones implementadas anteriormente y pasar de realizar 1 llamada a 16 llamadas para cada función. La otra alternativa es mantener las funciones implementadas como están y operar con la matriz compuesta de celdas salvo en el caso del cálculo de fuerzas para partículas con la misma posición. Esta solución, que a priori parece la única viable, fue desestimada tras su implementación. La solución planteada obliga a dividir la matriz de celdas en 16 sub-matrices dos veces por cada iteración (*ComputeDensities()* y *ComputeForces()*) de la aplicación y luego volver a unirlas. Este movimiento de datos supone una sobrecarga superior a la mejoría esperada con ArBB, lo que imposibilita, una vez más, la optimización de dichas funciones.

A modo resumen, en esta segunda versión se ha conseguido optimizar con ArBB las funciones en las que no se comparte información entre partículas, dejando únicamente sin paralelizar las funciones *ComputeDensities()* y *ComputeForces()*. Con el trabajo realizado hasta el momento se consiguen tiempos inferiores a la versión secuencial pero que podrían no ser suficientes. Así pues, es necesaria una tercera versión en la que se trate de resolver el problema de las funciones *ComputeDensities()* y *ComputeForces()* de forma definitiva.

4.2.4. Versión 3.0

Para esta tercera y última versión implementada de Fluidanimate se ha tomado el código con la puesta a punto realizada, las variables de ArBB utilizadas en la primera versión y las funciones *RebuildGrid()*, *InitAcelerationAndDensity()*, *ModifyDensity()*, *ProcessCollisions()* y *AdvanceParticles()* de la segunda versión implementada. El único trabajo que queda por realizar es la optimización de las dos funciones restantes, como se describió en el punto anterior, la paralelización haciendo uso de Intel ArBB no es factible así que, puesto que la compatibilidad entre ArBB y TBB parece viable, la solución al problema es utilizar Intel TBB de forma íntegra para estas dos funciones. La variación entre la solución planteada en la versión 2.0 no difiere mucho de esta, solo en la computación de fuerzas entre partículas con la misma posición en las celdas.

Mientras que Intel ArBB hace uso de los tipos propios de esta biblioteca, Intel TBB utilizará los arrays de C++ originales que contienen una copia dinámica del contenido de las variables de ArBB gracias a la función *bind()*. La utilización de tareas con TBB difiere del trabajo realizado hasta ahora, el cual estaba más enfocado a la paralelización de datos. La división del trabajo en esta nueva unidad de medida es clave para obtener un buen rendimiento. La utilización de una cola donde se inserte cada partícula que quiera calcular su densidad y aceleración como una tarea independiente no es eficiente,

[8] 16*16 – 16 interacciones entre partículas con la misma posición (ArBB).

ya que este cálculo necesita una inversión pequeña de tiempo y la utilización de una tarea por cada partícula supondría un grano demasiado fino, lo cual sería contraproducente. Así pues, una buena opción es la de asignar bloques de celdas a cada tarea, la dimensión de estos bloques variará en función del número de tareas, de esta forma el primer bloque de celdas corresponde con la primera tarea creada y así sucesivamente.

Tras decidir cuál es la división del trabajo, tarea que con ArBB no era necesaria, ahora es turno de escoger la estructura y el orden de ejecución de las tareas. La utilización de *task::enqueue,* incluido en la versión 3.0 de Intel TBB, podría ser interesante. El funcionamiento este tipo de cola es el siguiente: crear tareas e irlas introduciendo de forma dinámica en la cola para que sean ejecutadas cuando algún hilo esté ocioso. La sencillez de esta estructura la hacen ser una opción a tener en cuenta, no obstante *task::enqueue* ejecuta las tareas de forma automática cuando haya recursos disponibles por lo que no se puede garantizar que una tarea se lance en el momento deseado. La otra opción es utilizar una lista FIFO, la clase *tbb::task_list* es similar a *task::enqueue* pero en este caso si se le otorga el control al desarrollador para elegir cuando una tarea es lanzada y es posible jerarquizar las tareas, además en este caso la creación de tareas se realizará en un instante dado sin necesidad de crear ninguna a posteriori. La estructura *tbb::task_list* es más adecuada a las necesidades de la aplicación. Mediante el método *push_back()* se introducirán las tareas al final, una vez insertadas todas las tareas mediante el método *spawn()* se lanzará la ejecución de estas. Todas estas tareas dependen de una tarea denominada *root* que se encarga de crear la lista y esperar a que finalicen todas las tareas hijas. En cuanto a la jerarquía de tareas no es necesario identificar varios niveles puesto que todas las tareas, sin distinción, realizan la misma operación. En la Ilustración 12 se muestra la división del trabajo en bloques para un ejemplo compuesto por 12 celdas con 4 tareas.

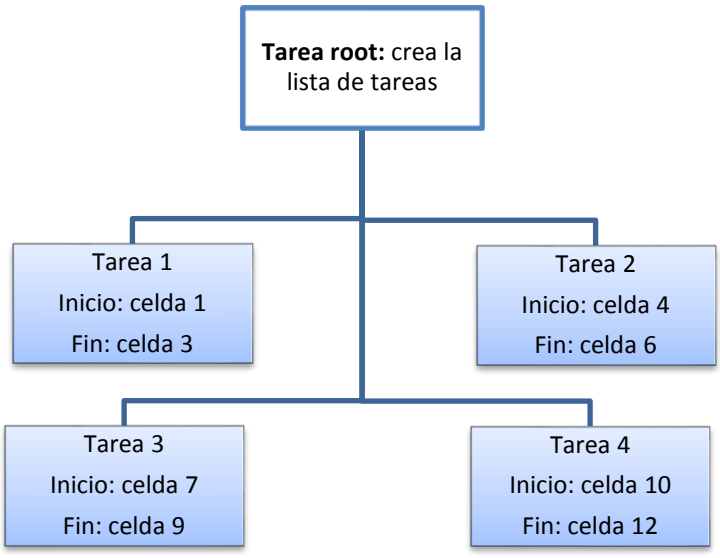


Ilustración 12. Ejemplo de la división del trabajo en tareas

El algoritmo de las funciones *ComputeDensities()* y *ComputeForces()* con TBB es similar al de la versión secuencial, las únicas variaciones reseñables son la utilización de arrays simples en lugar de las estructura *Cell* que incluía la versión secuencial y la operación parcial de la matriz por cada llamada debido a la división del trabajo en tareas. Las función implementadas con Intel TBB son funciones propias de C++ por lo que no hay que preocuparse por las restricciones que imponía ArBB como pueden ser la ejecución a la totalidad de la matriz o la utilización de variables propias de la librería.

Tras implementar ambas funciones hay aun trabajo que realizar, en este momento el código implementado es una copia del algoritmo original, pero este algoritmo realiza algunas operaciones que podrían ser modificadas con el fin mejorar la eficiencia de ambas funciones. En la versión secuencial, ambas funciones utiliza la función *GetNeighborCells()* que se encarga de devolver en un array las distintas posiciones de las celdas vecinas a partir de unas coordenadas dadas en los ejes x,y,z. La utilización de esta función no es necesaria y puede fusionarse con las propias funciones evitando la llamada a la función y la declaración y utilización de un array. De esta forma varía el orden de ejecución lo que permite explotar los principios de localidad de la memoria caché. En la versión secuencial primero se recorren las partículas de la celda con la que se está operando y después cada celda vecina, en esta versión el proceso es al contrario. La principal ventaja es que es más probable que se encuentre en la memoria caché la información de 16 partículas contiguas a la información de 27 celdas situadas en posiciones alejadas entre ellas, con lo cual es preferible recorrer las 16 partículas 27 veces que las 27 celdas 16 veces.

Otra optimización importante es la búsqueda eficiente de vecinos, en ambas funciones tras acceder a la partícula anfitriona y a la partícula vecina se comprueba que la partícula vecina sea inferior a la partícula anfitriona, ya sea porque esté situada en una celda inferior o porque tenga una posición menor dentro de la misma celda. Esto quiere decir que solo la mitad de los vecinos obtenidos en la búsqueda son realmente utilizados. En la Ilustración 13 se muestran de color verde las celdas utilizadas realmente y en rojo las que tras obtener su posición son desechadas.

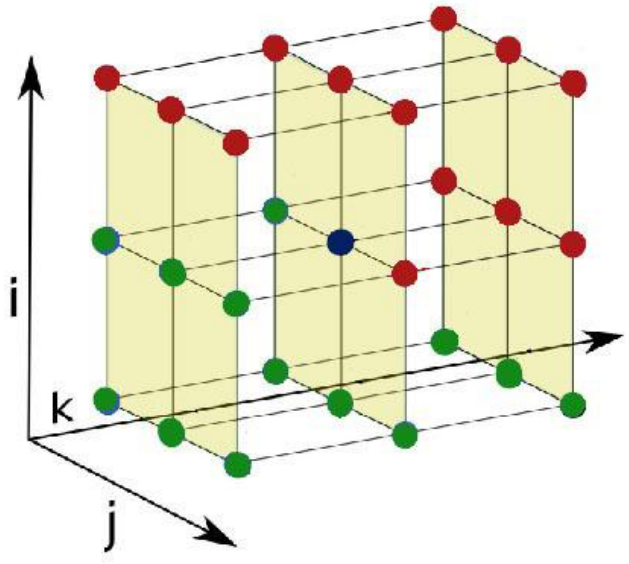


Ilustración 13. Búsqueda eficiente de vecinos

Esta búsqueda de celdas adyacentes supone un gran desperdicio del tiempo de cómputo, así pues las nuevas funciones solo accederán a los vecinos inferiores obviando el acceso al resto de celdas contiguas. Las celdas vecinas son bien conocidas, por lo que su búsqueda es predecible, aprovechando esta mejora se ha realizado un desenrollado del bucle original eliminado la exploración programática de celdas vecinas. Con esta última optimización finaliza la tarea de implementación de la aplicación Fluidanimate.

Una vez finalizada la tarea de programación es necesario evaluar si los resultados obtenidos entre la versión secuencial tomada como referencia y la nueva versión son los mismos. El análisis de las soluciones de ambos códigos se ha realizado a partir del fichero *in_35K.fluid* del benchmark PARSEC y en los resultados aparecen pequeñas diferencias resumidas en la Tabla 16. Como se describe en [20] las operaciones aritméticas en coma flotante no poseen una precisión absoluta, basándose en el redondeo. Este tipo de operaciones en Fluidanimate se realiza con número de precisión simple de 32 bits, de los cuales 23 son de mantisa, esto permite una precisión aproximada de 6 decimales. En la función *ComputeDensities()*, para el cálculo de la densidad de cada partícula, se realiza una suma de varios números muy pequeños, en la versión secuencial se sigue un orden predefinido, pero en la versión implementada el orden viene dado por el número de hilos, esto da lugar a que ambas operaciones devuelvan resultados distintos a partir del quinto decimal. La densidad es una variable que más tarde se utiliza para calcular el resto de atributos de una partícula, por lo que el error introducido tiene una correlación directa en el resto, no obstante las diferencias entre ambas versiones son ínfimas.

	Fichero in_35K
Número de partículas	35.402
Suma de errores	0,032221925
Error máximo	1E-05
Error mínimo	0
Promedio	1,75043E-06
% de diferencia entre ArBB y secuencial	0,000000008129%

Tabla 16. Errores introducidos en Fluidanimate con ArBB y TBB

Como ya se realizó en la versión optimizada de Blackscholes, es interesante consultar el informe de la herramienta Intel® VTune™ Amplifier XE de Intel que permite analizar el comportamiento de aplicaciones paralelizadas mediante los modelos TBB y ArBB y así comprobar que la implementación realizada hace uso de todos los recursos disponibles. El análisis se ha realizado con una entrada que simula una situación real con un total de 500 iteraciones que es el número de iteraciones por defecto, la fase de escritura al ser opcional no ha sido tenida en cuenta. El computador utilizado tiene un total de 8 núcleos (punto 5.1.1). Los resultados, tras el análisis de VTune, puede observarse en la **¡Error! No se encuentra el origen de la referencia.**. En esta ilustración parecen tres gráficos, el primero de ellos muestra el tiempo de cómputo de cada core, el segundo gráfico muestra el cómputo global de la ejecución y el último gráfico describe la utilización de varios cores de forma simultánea.

Los gráficos han sido divididos en 7 fases que se corresponden con las distintas funciones:

- **Fase de lectura (InitSim):** esta primera fase no ha sido paralelizada, se utiliza un único core con intervalos de alto rendimiento cercanos al 100% del uso de CPU.
- **RebuildGrid:** Primera función paralelizada, su complejidad no es excesiva, utiliza los 8 hilos de ejecución y el uso de CPU está en torno al 200%.
- **InitAcelerationAndDensity:** Esta función no necesita de mucho tiempo de cómputo y el comportamiento en la evaluación es similar al de la función anterior con un uso de CPU próximo al 200%.
- **ComputeDensities:** Primera de las funciones que posee una importe carga de trabajo y que ha sido paraleliza con TBB. Su rendimiento es excelente, y a la espera de los tiempos, se utilizan todos los recursos disponibles con una utilización cercana al máximo ideal del 800%.
- **ModifyDensity y ComputeForces:** La función *ModifyDensity* en la gráfica se fusiona con su sucesora *ComputeForces* debido al escaso tiempo que necesita en ser ejecutada. La función *ComputeForces*, paralelizada con TBB, se asemeja al comportamiento de *ComputeDensities* a mayor escala.
- **ProcessCollisions:** Esta función, paralelizada con ArBB, también consigue una buena utilización de los recursos, aproximándose al límite ideal del 800%.

- **AdvanceParticles:** La última función de la aplicación Fluidanimate hace uso de los 8 hilos de ejecución disponibles y por un tiempo reducido accede a la utilización de todos ellos de forma concurrente.

Además de las gráficas, VTune muestra una valoración acerca del uso de la CPU. En esta calificación un 60,9% lo considera excelente que se corresponde con las principales funciones implementadas, el 34,0% es calificado como pobre debido, en su mayoría, a la parte de lectura y un 4,8% como aceptable que se corresponde con aquellas funciones pequeñas que no aprovechan al máximo todos los recursos disponibles.

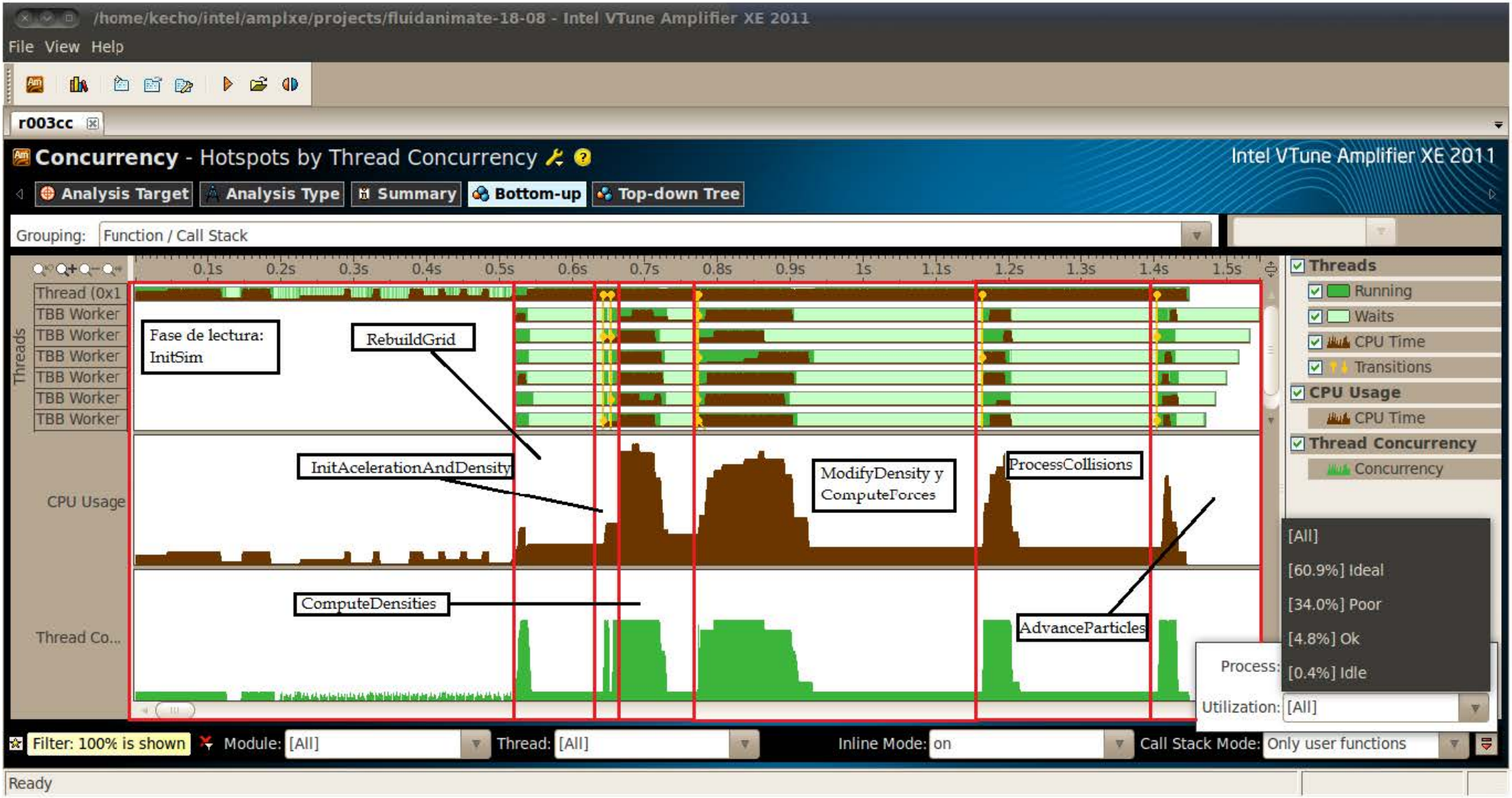


Ilustración 14. Resultados de la herramienta VTune para Fluidanimate 3.0

5. EVALUACIÓN DEL RENDIMIENTO

Dentro del presente capítulo se realiza un análisis de los resultados obtenidos tras la ejecución de ambas aplicaciones paralelizadas en distintas máquinas y con distintos ficheros de entrada. Una vez obtenidos los tiempos de medición se realiza una evaluación completa de los resultados que servirán para determinar si se han cumplido los objetivos planteados.

5.1.Hardware utilizado

La evaluación del rendimiento de ambas aplicaciones ha sido llevada a cabo utilizando dos máquinas. A continuación se muestra una descripción detallada de cada una de ellas:

5.1.1. Arquitectura unisocket con memoria uniforme

- **Procesador**: Intel Core i7 Q720
  - Arquitectura: Nehalem
  - Frecuencia: 1,6 GHz
    - Con la tecnología Turbo Boost[9] se alcanzan 2,8 GHz.
  - Número de núcleos: 4
    - Con la tecnología Hyper-Threading se consiguen 8 hilos.
  - Tamaño de memoria caché L3: 6MB
  - Extensiones del conjunto de instrucciones: SSE 4.2
- **Memoria RAM**: 4,00 GB
- **Sistema Operativo:** Linux (Ubuntu 10.04)

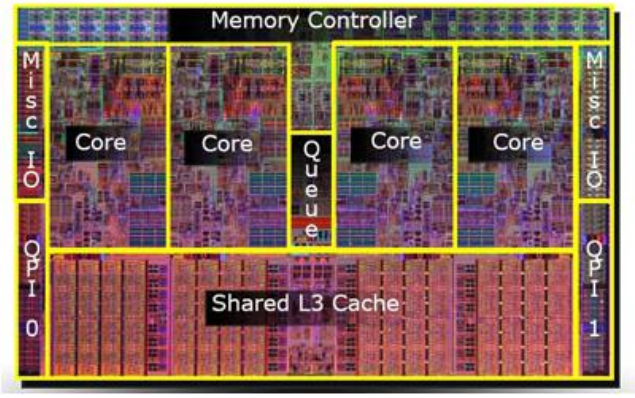


Ilustración 15. Ejemplo de procesador de la familia Core i7 con 4 cores

[9] Permite el aumento de la frecuencia de los procesadores en caso de carga de trabajo elevada.

5.1.2. Arquitectura multi-socket con memoria no uniforme

- **Procesador:** 4 procesadores Intel Xeon E7 - 4807
  - Arquitectura: Nehalem
  - Frecuencia: 1,87 GHz
    - No posee la tecnología Turbo Boost.
  - Número total de núcleos: 24 (6 núcleos por procesador)
    - Con la tecnología Hyper-Threading se consiguen 48 hilos.
  - Tamaño de memoria caché L3: 18MB
  - Extensiones del conjunto de instrucciones: SSE 4.2
- **Memoria RAM:** 128,00 GB
- **Sistema Operativo:** Linux

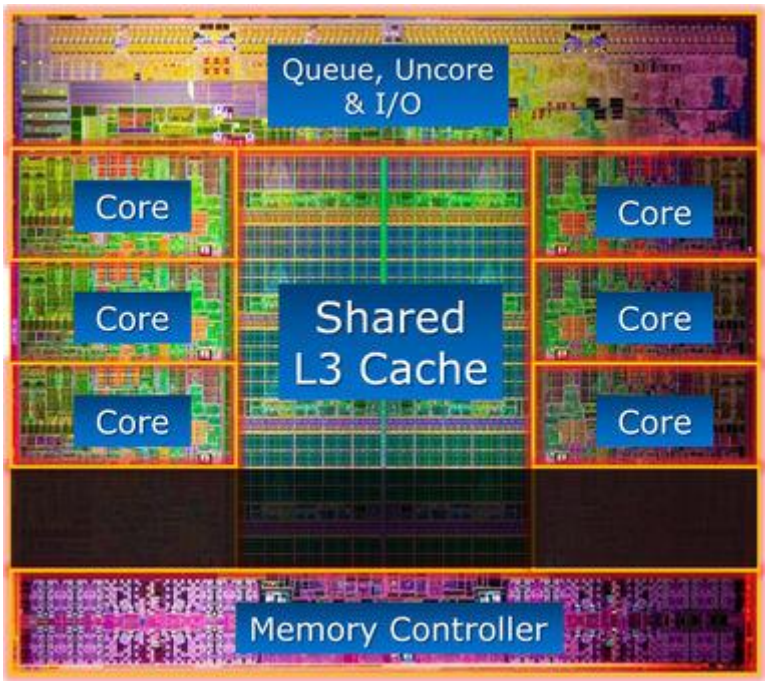


Ilustración 16. Ejemplo de procesador de la familia Xeon E7 con 6 cores

Especificación CPU	Sockets	Cores por socket	Hyperthreading	Turbo Boost	Memoria (GB)	Tipo de memoria
Intel® Core™ i7 CPU Q720 @ 1.60GHz	1	4	Sí	Sí (2,80 GHz)	4	UMA
Intel® Xeon® CPU E7-4807 @ 1.87GHz	4	6	Sí	No	128	ccNUMA

Tabla 17. Arquitecturas usadas en la evaluación

5.2.Metodología

La toma de tiempos se ha realizado usando la función *gettimeofday()* de la biblioteca *sys.time.h* del lenguaje C. Para ello, basta con tomar la hora del sistema al iniciar la ejecución del programa, al finalizar la ejecución y calcular la diferencia entre ambas. No obstante, para garantizar la fiabilidad de los tiempos obtenidos y corroborarlos, se ha utilizado el mandato *time* del sistema operativo Linux que muestra, entre otros, el tiempo transcurrido durante la ejecución del programa.

La toma de tiempos para cada caso de estudio se repite hasta que la desviación estándar sea irrelevante. El tiempo incluido en los resultados es la media de estos valores. La aceleración se calcula tomando como base el tiempo de ejecución de la versión secuencial incluida en el benchmark PARSEC.

5.3.Casos de estudio

En este punto se exponen los diferentes ficheros de entrada para cada una de las aplicaciones paralelizadas. Todos los ficheros utilizados se corresponden con entradas originales del benchmark PARSEC.

5.3.1. Blackscholes

Los ficheros de entrada de la aplicación Blackscholes, contienen un número determinado de opciones financieras. Cada opción, contiene la información necesaria para poder calcular el precio estimado resultante. Para la evaluación del rendimiento se ha optado por seleccionar ficheros con una alta carga computacional, lo cual permitirá observar de forma perceptible las diferencias entre las distintas tecnologías. Así pues, las pruebas realizadas se componen de los siguientes casos de estudio:

- **Cartera con un número moderado de opciones**
  - Número de opciones: 65.536
  - Tamaño: 3,94 MB
  - Tipo de entrada en PARSEC: *simlarge*

- **Cartera con un número grande de opciones**
  - Número de opciones: 10.000.000
  - Tamaño: 602 MB
  - Tipo de entrada en PARSEC: *native*

5.3.2. Fluidanimate

Los ficheros de entrada de la aplicación Fluidanimate contienen un número determinado de partículas. Cada partícula contiene la información necesaria (aceleración, posición, velocidad y viscosidad) para poder realizar la simulación del fluido. Para evaluar el rendimiento de los distintos modelos de programación paralela se ha optado por seleccionar ficheros de diferentes tamaños para analizar cuál es el impacto de ArBB según la carga computacional del caso de estudio. Así pues, las pruebas realizadas se componen de los siguientes casos de estudio:

- **Conjunto pequeño de partículas**
  - Número de partículas: 35.402
  - Dimensión de la matriz: 30x41x30
  - Tamaño: 1,25 MB
  - Tipo de entrada en PARSEC: *simsmall*

- **Conjunto moderado de partículas**
  - Número de partículas: 102.850
  - Dimensión de la matriz: 32x45x32
  - Tamaño: 3,53 MB
  - Tipo de entrada en PARSEC: *simmedium*

- **Conjunto grande de partículas**
  - Número de partículas: 305.809
  - Dimensión de la matriz: 46x64x46
  - Tamaño: 10,4 MB
  - Tipo de entrada en PARSEC: *simlarge*

- **Conjunto enorme de partículas**
  - Número de partículas: 501.642
  - Dimensión de la matriz:55x76x55
  - Tamaño: 17,2 MB
  - Tipo de entrada en PARSEC: *native*

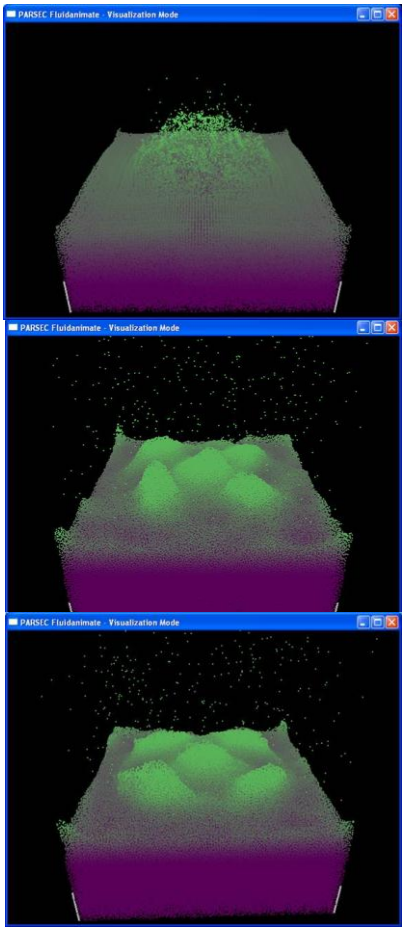


Ilustración 17. Modo de visualización en Fluidanimate

5.4.Resultados

En esta sección se muestran los tiempos de ejecución de cada uno de los casos de estudio. Para poder evaluar el trabajo realizado se han efectuado pruebas haciendo uso de la ley de Amdahl, la fórmula para obtener la aceleración máxima, la ecuación para obtener la eficiencia del hardware utilizado y las versiones de las aplicaciones paralelizadas con otros modelos.

Antes de comenzar la evaluación, recordar que la versión de ArBB para Blackscholes está muy orientada a la utilización del modelo SIMD lo que generará una mejor optimización, por el contrario esta versión peca de una alta tasa de fallos de caché que, como se describirá más adelante, puede ser un factor decisivo para la aceleración de una aplicación. La versión de ArBB y TBB para Fluidanimate está más enfocada a la utilización del paralelismo multicore debido a la inclusión de TBB y por ende la pérdida de protagonismo de ArBB, sin embargo esto permite que la tasa de fallos de caché será menor, ya que TBB realiza una política de planificación eficiente de la memoria.

5.4.1. Blackscholes

5.4.1.1. Cartera con un número moderado de opciones

La primera toma de tiempos se ha realizado sobre el fichero *simlarge*, compuesto por 65536 opciones, con el número de repeticiones que viene por defecto en el benchmark PARSEC, 100. Como se puede apreciar en la Tabla 18, la interferencia de la parte secuencial es de en torno a un 17% para ambas arquitecturas, esto supone que la aceleración conseguida se vea lastrada por las fases de lectura y escritura secuenciales.

	Computador Intel Core i7	Computador Intel Xeon E7
Fase de lectura	0,157 (10.37%)	0,220 (9.40%)
Fase de cómputo	1,261 (83.07%)	1,941 (82.61%)
Fase de escritura	0,099 (6.56%)	0,187 (7.99%)
Total	1,519 segundos	2,350 segundos

Tabla 18. Blackscholes: Tiempo secuencial por fases para 65.536 opciones

Arquitectura unisocket con memoria uniforme

El porcentaje del tiempo invertido en la fase secuencial asciende a 16.93% en el computador Intel Core i7. Una vez calculado esta información se puede obtener el límite teórico propuesto por Amdahl para la aceleración de la versión paralelizada con ArBB. Según la formula enunciada en el punto 2.2, la aceleración ideal es de 3,66. Esto implica que la versión implementada podría llegar a tener un tiempo 3,66 veces menor al tiempo de la versión secuencial con la utilización de los 8 hilos (Ecuación 1).

$$\frac{1}{(1-0.8307)+\frac{0.8307}{8}} = 3.66$$

Ecuación 1. Aceleración ideal para simlarge (Intel Core i7)

En los tiempos medidos, se obtiene una aceleración máxima de 3,60 para el procesador Intel Core i7 con 8 hilos, como puede observarse en el Gráfico 1.

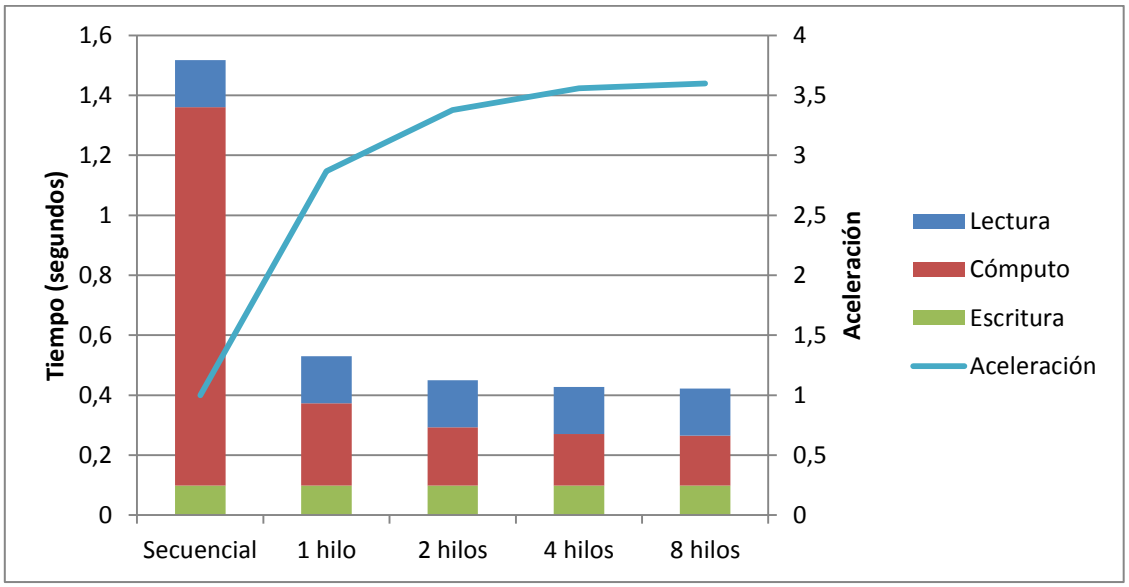


Gráfico 1. Blackscholes: Tiempos Intel Core i7 para simlarge

Pese a que la versión paralelizada con ArBB no supera el límite teórico de la ley de Amdahl, esta ley no es aplicable a los resultados obtenidos. El motivo se debe a que Gene Amdahl no tuvo en cuenta la paralelización vectorial que se consigue con los juegos de instrucciones SSE y AVX. Según la ley de Amdahl la aceleración máxima para un único hilo es uno, puesto que no se divide el trabajo entre varios cores. No obstante ArBB introduce una aceleración de 2,88 para un único hilo debido a la vectorización. En esta prueba se aprecia cómo la pendiente de la aceleración va disminuyendo aproximándose a cero, lo cual da a entender que un mayor uso de hilos podría llegar a ser contraproducente.

Arquitectura multi-socket con memoria no uniforme

La aceleración obtenida en la arquitectura ccNUMA es de 3,25 con 8 hilos. A partir de estos resultados pueden obtenerse conclusiones muy interesantes (Gráfico 2).

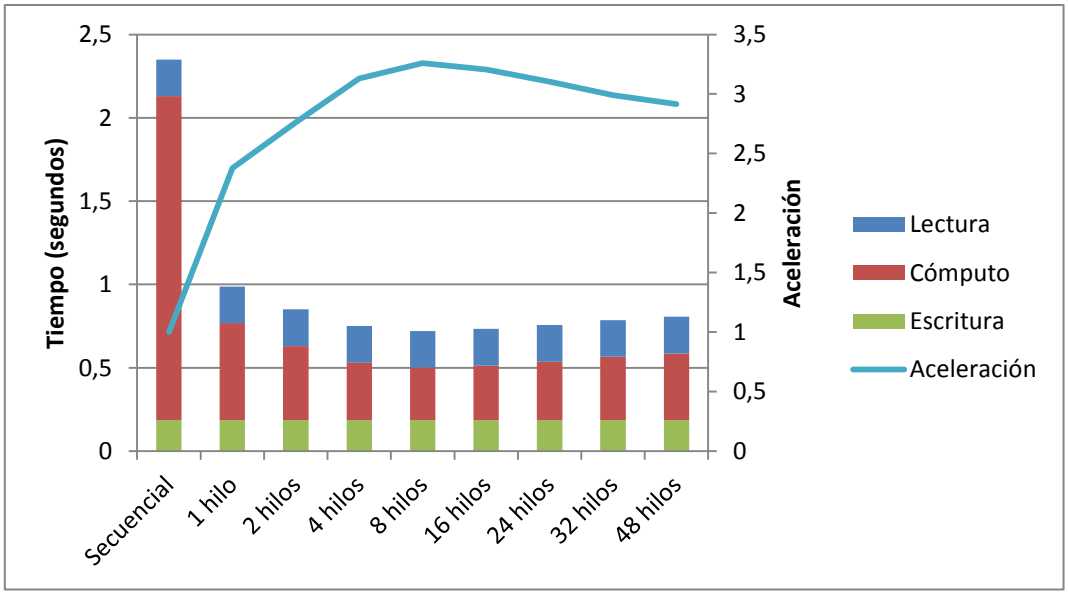


Gráfico 2. Blackscholes: Tiempos Intel Xeon E7 para simlarge

La primera de ellas es acerca de la relación entre el caso de estudio utilizado y la arquitectura. Cómo ya se presagiaba en el Gráfico 1, la aceleración crece hasta los 8 hilos y decrece a partir de ese momento. La utilización de más cores no implica una mejor aceleración, si la carga computacional no es muy grande y se utilizan más cores de los necesarios, el grano de trabajo es muy fino y hay demasiada sobrecarga. Para este caso de estudio, donde la carga de trabajo podría catalogarse como media, la utilización de tantos hilos de ejecución es inefectiva para el rendimiento, de ahí que los mejores resultados se obtengan con 8 de los 48 hilos disponibles.

Otro dato importante es la pérdida de rendimiento en este tipo de arquitecturas en comparación con la arquitectura UMA, siendo de 3,25 y 3,66 respectivamente. Las arquitecturas ccNUMA permiten incorporar un gran número de procesadores pero introducen penalizaciones en los fallos de caché. ArBB necesita gran cantidad de memoria[10] para poder optimizar el rendimiento de la aplicación, esto implica muchas operaciones de lectura y escritura en memoria y por ende más fallos de caché.

5.4.1.2. Cartera con un número grande de opciones

En este caso de estudio se computan 10.000.000 de opciones utilizando el fichero *native*. Con el fin de comprobar los límites de la versión implementada con ArBB se reducirán la interferencia secuencial realizan 500 iteraciones en lugar de 100. En la Tabla 19. puede verse esa reducción de la parte secuencial de la aplicación y los tiempos tomados como base en la versión secuencial.

[10] La utilización de ArBB implica un consumo de más de 150MB, sin contabilizar la memoria necesaria para la ejecución del programa.

	Computador Intel Core i7	Computador Intel Xeon E7
Fase de lectura	18,695 (1.76%)	24.46 (1.59%)
Fase de cómputo	1034.626 (97.42%)	1496.776 (97.76%)
Fase de escritura	8.708 (0.82%)	9.99 (0.65%)
Total	1062,027 segundos	1531.226 segundos

Tabla 19. Blackscholes: Tiempo secuencial por fases para native

Arquitectura unisocket con memoria uniforme

El porcentaje de la parte secuencial para este caso de estudio es de 2.58% respecto al tiempo total en lugar del 17% del caso anterior, esto permite que ArBB pueda paralelizar la mayor parte del tiempo de ejecución de Blackscholes.

En el Gráfico 3 pueden observarse los resultados obtenidos tras la medición de tiempos con una aceleración resultante de 13,40 para 8 hilos de ejecución.

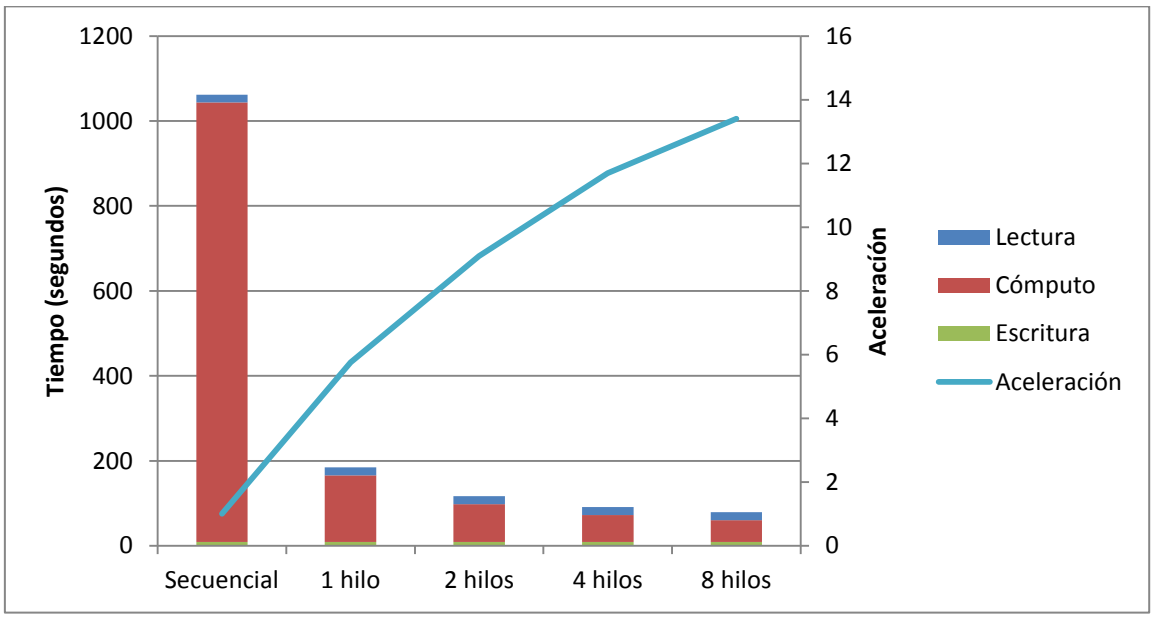


Gráfico 3. Blackscholes: Tiempos Intel Core i7 para native

Utilizando la Ley de Amdahl se obtiene una aceleración máxima para dichos tiempos de 6.78 (Ecuación 2).

$$\frac{1}{(1-0.9742)+\frac{0.9742}{8}} = 6{,}78$$

Ecuación 2. Aceleración ideal para native (Intel Core i7)

La aceleración máxima obtenida es muy superior a la resultante de la ecuación de Amdahl, lo que clarifica que la Ley de Amdahl no es aplicable para este modelo de

programación paralela. Por este motivo, esta no será tenida en cuenta para los casos de estudio posteriores.

En un caso de estudio donde la parte secuencial es ínfima y la carga computacional es lo suficientemente grande como para aprovechar los 8 hilos de manera eficiente el incremento de aceleración se mantiene relativamente proporcional al número de hilos, consiguiendo aceleraciones muy positivas.

Arquitectura multi-socket con memoria no uniforme

En esta prueba donde se cuenta con un gran número de opciones y un importante número de cores la aceleración crece hasta llegar a 28,86 con 48 hilos de ejecución (Gráfico 4).

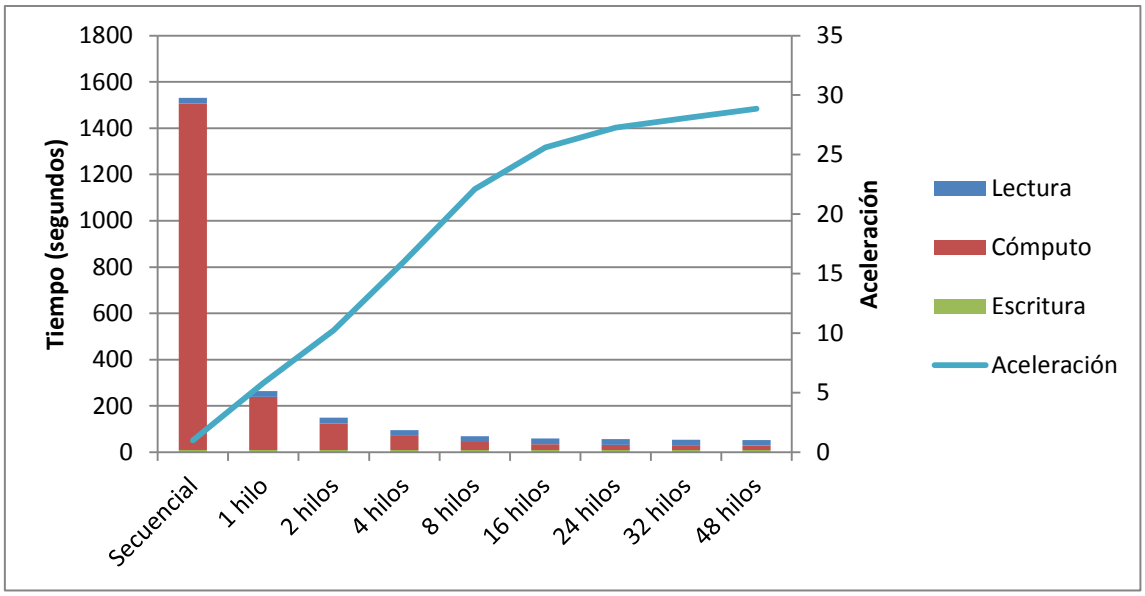


Gráfico 4. Blackscholes: Tiempos Intel Xeon E7 para native

En este caso, la aceleración es mucho mejor respecto a la arquitectura anterior puesto que si se pueden aprovechar los 48 hilos de ejecución disponibles. Pese a la penalización que introduce la arquitectura ccNUMA en los fallos de caché, ArBB consigue una aceleración muy buena.

Sin embargo, remarcar que a pesar del buen aprovechamiento de los recursos en esta prueba, con 16 hilos se aprecia un cambio de tendencia en la aceleración, reduciéndose el crecimiento de esta de manera notable.

5.4.1.3. Comparación entre las arquitecturas

Después de analizar los tiempos obtenidos para los casos de estudio *simlarge* y *native* es interesante comprobar la eficiencia de la aplicación acelerada en cada arquitectura. Para el cálculo de este dato se hace uso de la ecuación mostrada en el punto 2.2 para el cálculo de la aceleración por core.

En el Gráfico 5 se muestran las aceleraciones por core para cada prueba realizada. Para la arquitectura UMA el número de hilos son 8 y para la arquitectura NUMA el número de hilos es 48. La eficiencia en el Intel Core i7 es de 0,44 y 1,67 para los casos de estudio *simlarge* y *native*, para el Intel Xeon E7 la eficiencia es de 0,06 y 0,6 respectivamente. Merece especial atención la eficiencia del caso de estudio *native* con el Intel Core i7 puesto que obtiene una aceleración superior a uno, cifra inalcanzable para los tradicionales modelos de paralelización.

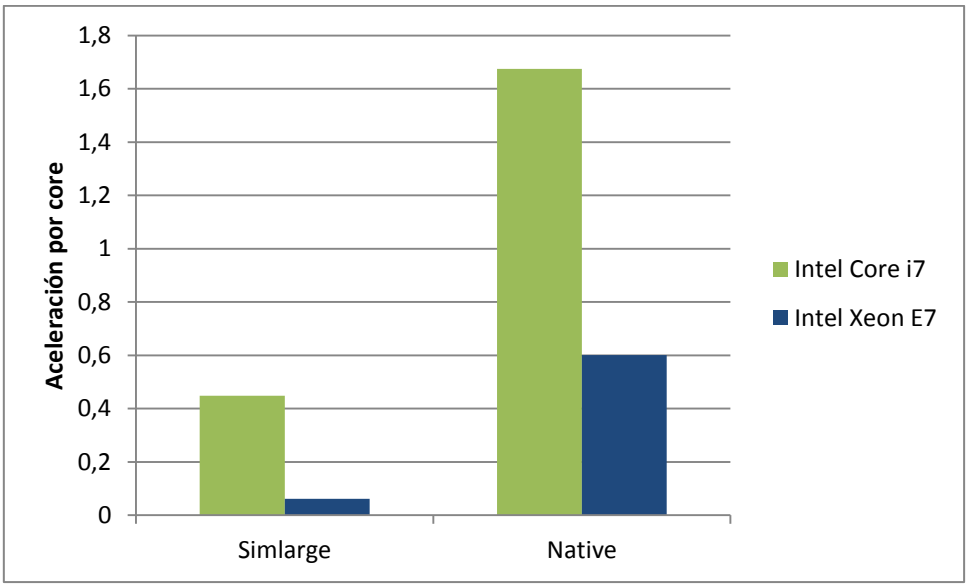


Gráfico 5. Blackscholes: Evaluación de la eficiencia

Los resultados obtenidos son similares para ambos ficheros, aunque a diferente escala. En ellos se proclama como claro vencedor el computador con el procesador Intel Core i7. Pese a ser el mismo código fuente para ambas arquitecturas, el Intel Core i7 cuenta con ventaja debido a dos factores, el más importante es la utilización de un modelo de memoria uniforme lo que hace menos vulnerable a los fallos de caché, el segundo dato a tener en cuenta es la mayor frecuencia del reloj siendo 2,8 GHz para el Intel Core i 7 y de 1,87 GHz para los Intel Xeon E7. No obstante la frecuencia del procesador Intel Core i7 no puede ser tomado como referencia absoluta puesto que utiliza la tecnología *Turbo Boost* que varía la frecuencia del procesador en función de la demanda.

5.4.1.4. Comparación con el resto de modelos de programación paralela

Por último, queda comprobar el rendimiento de ArBB con el resto de modelos de programación paralela. Estos modelos son diferentes entre sí, mientras que ArBB utiliza paralelismo a nivel de datos (DLP), el resto utiliza paralelismo a nivel de hilos

(TLP). No obstante, es interesante comprobar el rendimiento de los distintos modelos para la misma aplicación.

En el Gráfico 6 se observa la aceleración obtenida para 65.536 opciones (fichero *simlarge*) con 100 iteraciones, el incremento en la aceleración con ArBB es menos pronunciado que el resto debido a la utilización de la vectorización contando con un único hilo. Pese a esto, ArBB consigue sacar más de un punto de ventaja de aceleración respecto al resto. Los otros tres modelos poseen tiempos muy similares.

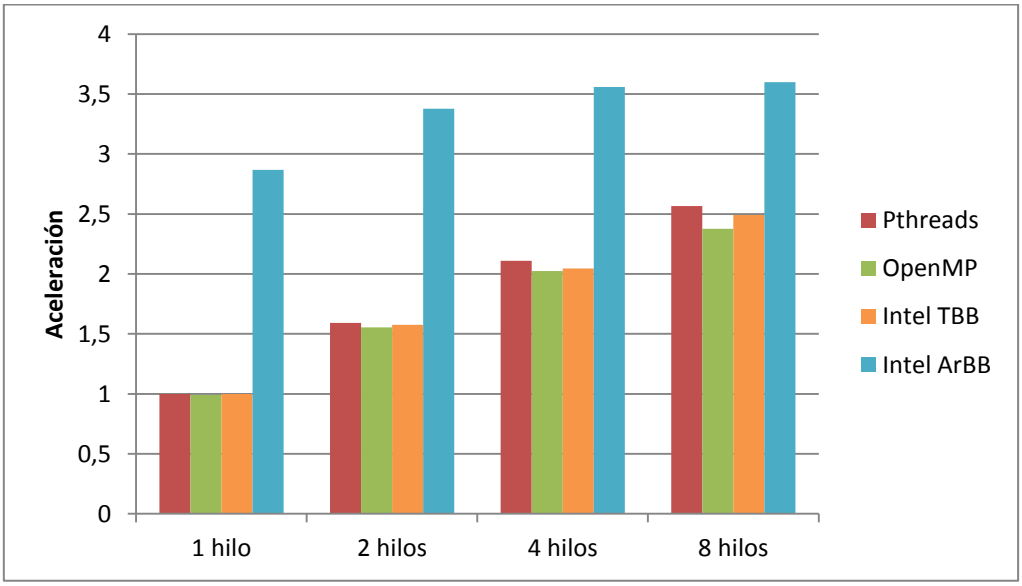


Gráfico 6. Blackscholes: Comparación Intel Core i7 para simlarge

A continuación, en el Gráfico 7, se observa el comportamiento de ArBB y el resto de tecnologías en una arquitectura ccNUMA. De ella pueden obtenerse dos datos interesantes. El primero de ellos es la pérdida de rendimiento de ArBB respecto Pthreads y TBB. Cómo se mencionó en el punto 5.4.1.1, debido a la escasa carga de trabajo, ArBB solo consigue aprovechar 8 hilos de los 48 disponibles, sin embargo, el resto llegan a aprovechar 24 hilos. Otro dato importante es la caída de rendimiento, tanto para ArBB como para OpenMP, debido a la penalización que introduce la arquitectura ccNUMA para los fallos de caché. A diferencia de TBB, OpenMP no explota los principios de localidad en la memoria caché ni sigue una planificación basada en reducir esta tasa de fallos, esto supone que OpenMP obtenga malos resultados.

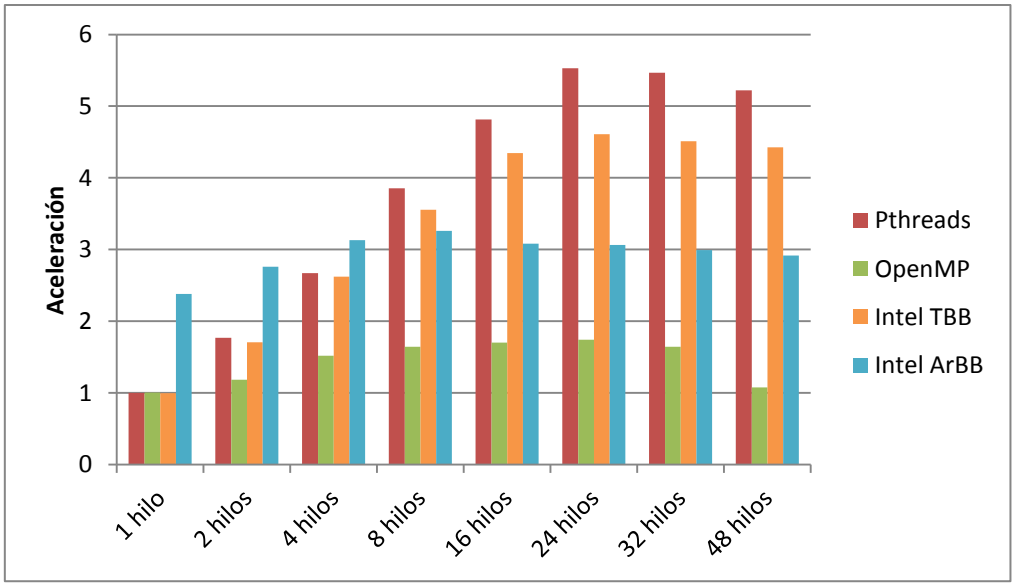


Gráfico 7. Blackscholes: Comparación Intel Xeon E7 para simlarge

Centrándose en el fichero *native*, de mayor tamaño y con 500 iteraciones, se aprecia en el Gráfico 8 como en la arquitectura unisocket con memoria UMA, ArBB saca una gran ventaja a sus competidores. Mientras que los tiempos de Pthreads, OpenMP e Intel TBB son similares, ArBB obtiene una ventaja de más de 10 puntos.

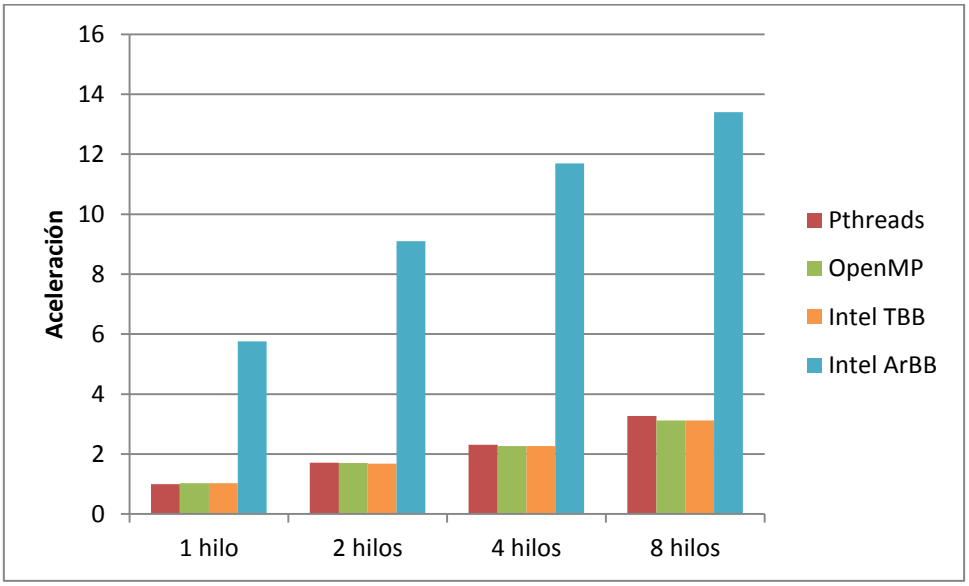


Gráfico 8. Blackscholes: Comparación Intel Core i7 para native

Mientras que con el fichero de 65.536 opciones ArBB no era eficiente, en esta prueba sí consigue obtener un mejor rendimiento que sus rivales. En el Gráfico 9, se observa cómo continúa la tendencia negativa de OpenMP debido a la alta tasa de fallos de

caché y cómo se mantiene esa diferencia de 10 puntos entre la aceleración de ArBB y Pthreads/TBB.

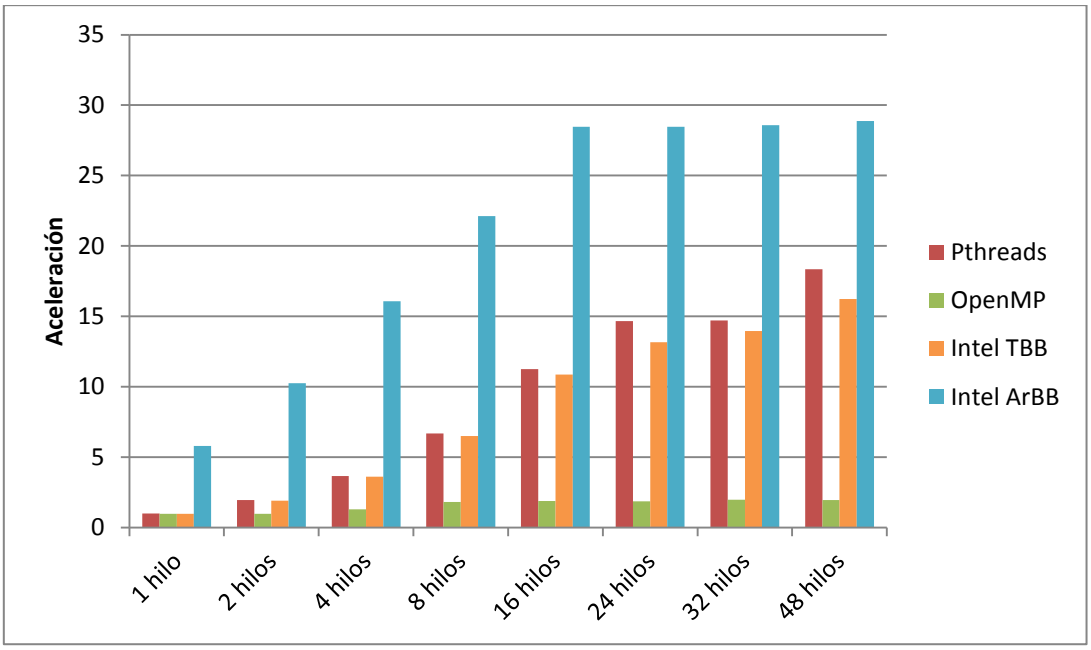


Gráfico 9. Blackscholes: Comparación Intel Xeon E7 para native

Finalmente, y a modo de resumen en el Gráfico 10 se muestran las aceleraciones para los distintos modelos modificando el número de iteraciones. Gracias a esta ilustración puede obtenerse una visión global de la información relevante obtenida anteriormente.

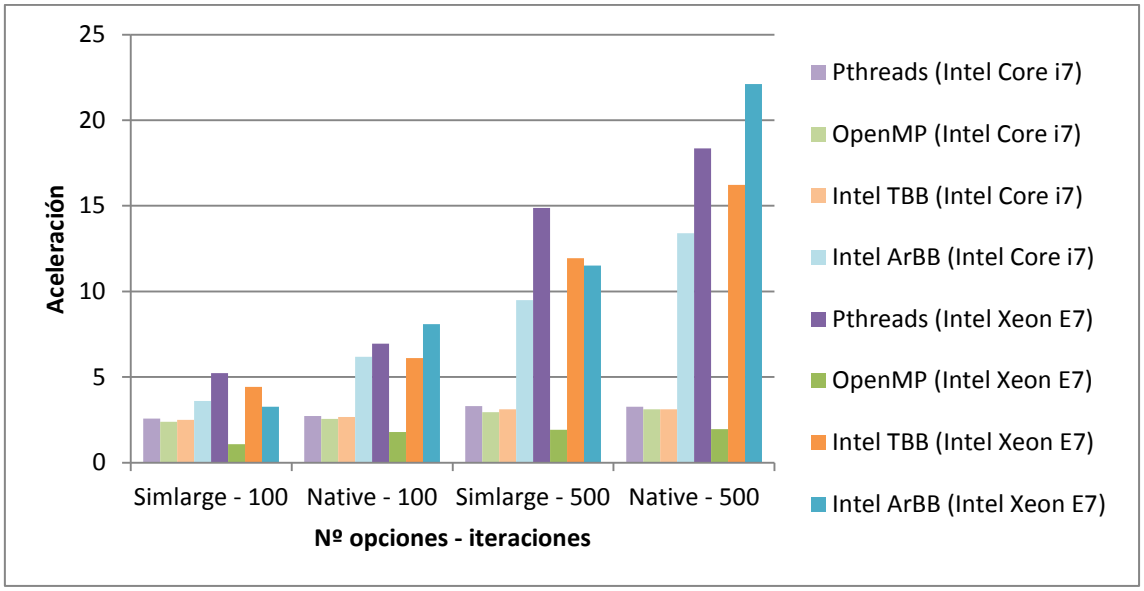


Gráfico 10. Blackscholes: Resumen comparativo

5.4.2. Fluidanimate

5.4.2.1. Conjuntos medianos de partículas

Con el objetivo de compactar la información, la primera toma de tiempos se ha realizado sobre los conjuntos de 35.000 partículas (*simsmall*) y 100.000 partículas (*simmedium*), que podrían ser considerados como ficheros de tamaño medio. El número de iteraciones realizadas es de 500, que es el número de iteraciones por defecto para algunas entradas de Fluidanimate dentro del benchmark PARSEC, además con un alto número de iteraciones se elimina cualquier influencia de la fase secuencial de lectura. La fase de escritura, al ser opcional, no ha sido tenida en cuenta para esta medición. Al no haber una participación perceptible de la fase secuencial de la aplicación, no se han divido los tiempos obtenidos en cada fase como se realizó en Blackscholes.

Arquitectura unisocket con memoria uniforme

Los primeros tiempos de la nueva versión implementada con ArBB y TBB arrojan resultados aceptables. La aceleración es de 2,1 y 2,7 para los ficheros *simsmall* y *simmedium* respectivamente. La aceleración obtenida con el fichero *simsmall* clarifica que este tipo de ficheros no permiten un gran rendimiento debido a dos factores importantes. El primero de ellos es la dimensión de la matriz, siendo esta de un tamaño medio, ya que la carga de trabajo no es muy elevada. El segundo factor importante es la baja ocupación de la matriz, solo un 5,99% de partículas existen realmente, a pesar de que ArBB computa el 100% de las partículas. Fruto de estos factores es la baja aceleración obtenida, llegando a ser inferior a uno (0,97) para un único hilo.

Si se echa un vistazo atrás, el rendimiento de ArBB con Blackscholes en este tipo de arquitecturas era bien distinto, con un único hilo se adquirían aceleraciones satisfactorias superiores a 1. Esto se debe, en gran medida, a la fusión de ArBB con TBB, que implica la pérdida de protagonismo de la paralelización SIMD en detrimento de la paralelización con hilos. Como resultado, la inclusión de más hilos de ejecución permite un gran aumento de la aceleración y por tanto del rendimiento.

El fichero *simmedium* arroja mejores resultados que el fichero *simsmall*, esto se debe a su mayor porcentaje de ocupación de las partículas y el aumento del tamaño de la matriz tridimensional. En este fichero si se aprecia el impacto de la paralelización SIMD que realiza ArBB, consiguiendo una aceleración de 1,14 con un único hilo. Toda la información descrita está ilustrada en el Gráfico 11.

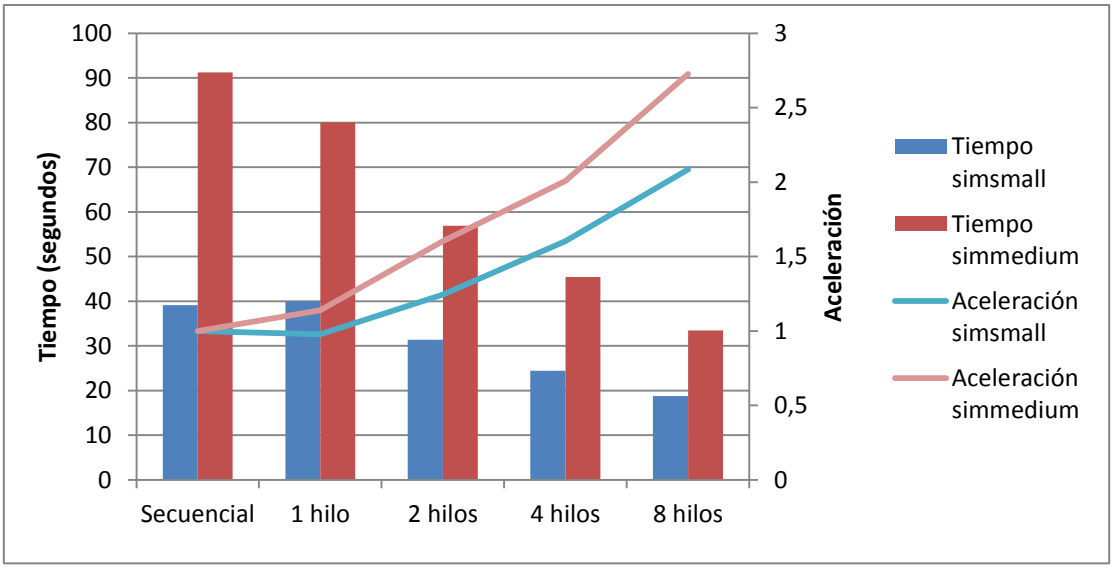


Gráfico 11. Fluidanimate: Tiempos Intel Core i7 para simsmall y simmedium.

Arquitectura multi-socket con memoria no uniforme

En el Gráfico 12, se muestran los tiempos obtenidos para los ficheros *simsmall* y *simmedium* en la arquitectura NUMA. Las aceleraciones conseguidas son de 5,3 y 8,4 para los ficheros *simsmall* y *simmedium* respectivamente. La aceleración máxima se consigue con 32 hilos en lugar de los 48 disponibles debido a la escasa carga computacional de los casos de estudio y a la reducida granularidad del trabajo a realizar. En esta arquitectura, las diferencias entre ambos ficheros son más notables que en el caso anterior, ya que se alcanza el límite de optimización de ambos casos de estudio, donde el límite del fichero *simsmall* es bastante menor que el del fichero *simlarge*. No obstante, las aceleraciones obtenidas son aceptables para ambos casos.

Un dato muy interesante es la similitud entre ambas arquitecturas, mientras que en Blackscholes el rendimiento de la aplicación con ArBB caía en picada con la arquitectura NUMA, en Fluidanimate las aceleraciones son muy similares en ambos escenarios. Este nuevo comportamiento, se debe, en gran medida, a la utilización de Intel TBB en la aplicación. Mientras que ArBB introduce una alta tasa de fallos de caché, Intel TBB vela por reducir este porcentaje. Gracias a esta fusión que evita la penalización de tiempo que se produce en arquitecturas NUMA con una alta tasa de fallos de caché se consigue un buen rendimiento, al contrario que en Blackscholes que únicamente utilizaba Array Building Blocks.

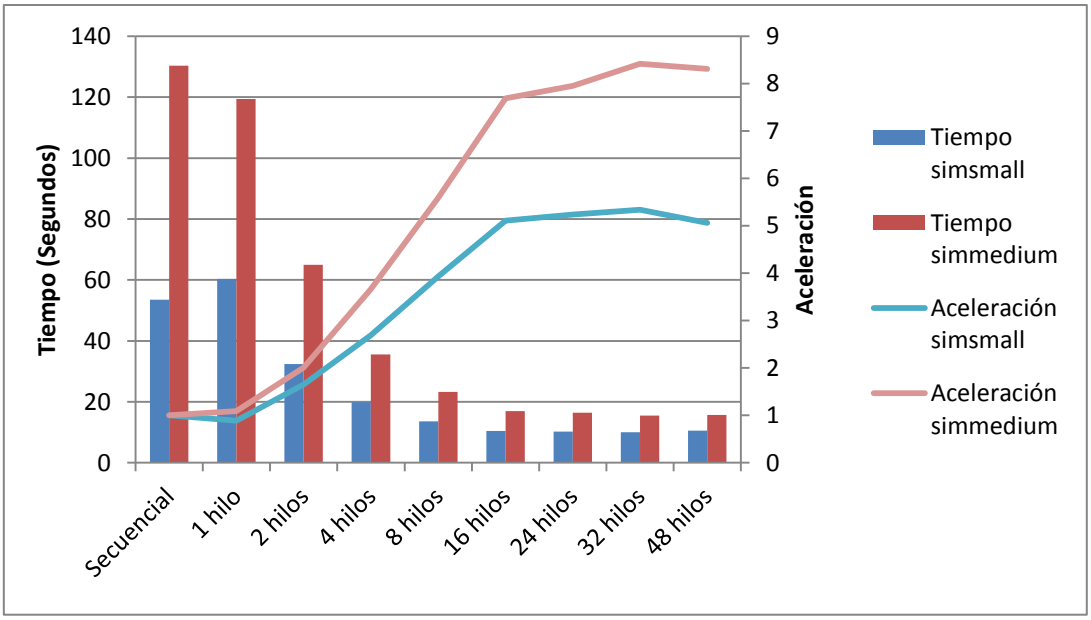


Gráfico 12. Fluidanimate: Tiempos Intel Xeon E7 para simsmall y simmedium

5.4.2.2. Conjuntos grandes de partículas

En esta medición se muestran los resultados para ficheros de gran tamaño, como son los ficheros *simlarge* y *native*. El número de iteraciones es de 500, al igual que el caso anterior, ya que es el número de iteraciones establecido por defecto. La fase de escritura tampoco es tenida en cuenta en esta prueba y por lo tanto no se realiza la división en distintas fases.

Arquitectura unisocket con memoria uniforme

En esta primera prueba para ficheros de gran tamaño se obtienen aceleraciones próximas a 3 para ambos ficheros (Gráfico 13). Las aceleraciones obtenidas no son mucho mejores que los casos de estudio de menor tamaño, en gran medida debido a la escasa ocupación de partículas en estos ficheros también. Para el fichero *simlarge*, el porcentaje de ocupación real es del 14,11% y para el fichero *native* el porcentaje se sitúa en 13,63%. Ambas aceleraciones se asemejan mucho, ya que pese a ser *native* algo más grande los niveles de ocupación del fichero es similar.

Otro dato interesante que arroja esta prueba es la efectividad de ArBB con un único hilo, mediante la utilización del juego de instrucciones SSE, donde la aceleración empieza a adquirir protagonismo, situándose cerca del 1,3 para ambos ficheros.

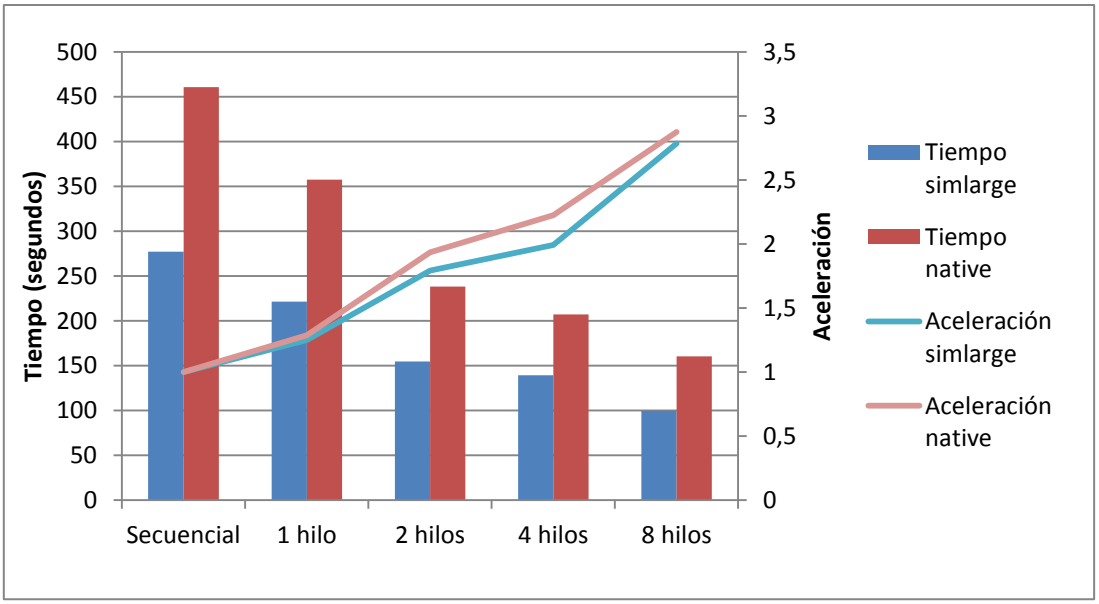


Gráfico 13. Fluidanimate: Tiempos Intel Core i7 para simlarge y native

Arquitectura multi-socket con memoria no uniforme

En la arquitectura NUMA, los ficheros de gran tamaño obtienen aceleraciones de 8,1 y 8,5 para los ficheros *simlarge* y *native* respectivamente. En el Gráfico 14 se pueden observar los tiempos obtenidos. En dichos tiempos se extraen tres datos interesantes, el primero de ellos es el aprovechamiento de los 48 hilos disponibles, lo que cual evidencia que estos casos de estudio se corresponden con ficheros de una alta demanda computacional. El segundo de los datos a resaltar es la similitud a la arquitectura UMA debido a la utilización de TBB. Por último, y no menos importante, las aceleraciones obtenidas con 48 hilos pueden calificarse como muy positivas, ya que se está obteniendo una reducción de más de una octava parte del tiempo secuencial.

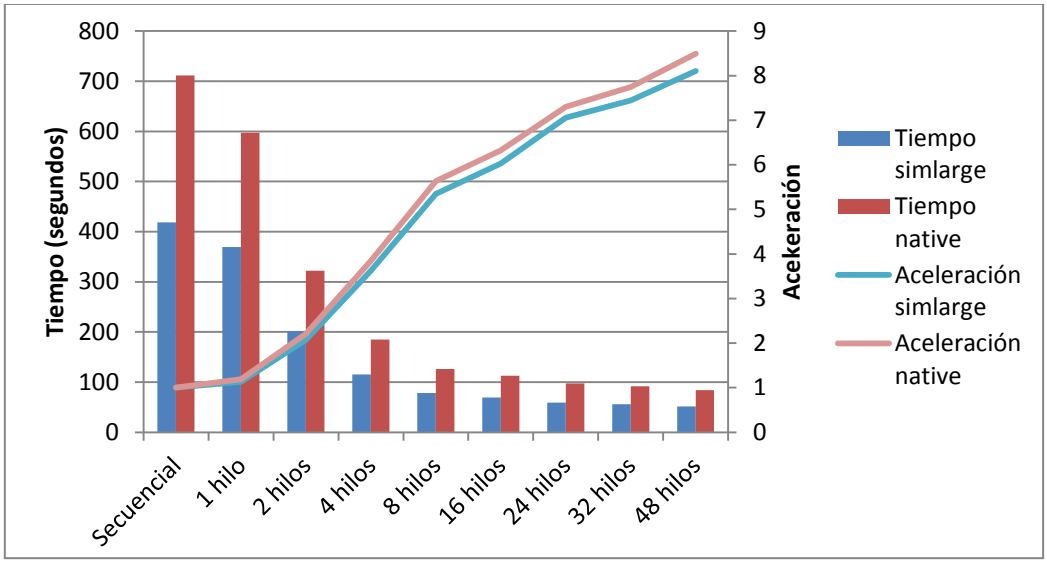


Gráfico 14. Fluidanimate: Tiempos Intel Xeon E7 para simlarge y native

5.4.2.3. Conjuntos con un alto nivel de ocupación de partículas

Tras las pruebas realizadas se evidencia la efectividad de ArBB junto con TBB en Fluidanimate. No obstante, ArBB ha tenido que lidiar con una baja tasa de partículas durante todos los casos de estudio, lo que provocaba unos resultados por debajo de las capacidades de este nuevo modelo de programación paralela. Así pues, es interesante comprobar el rendimiento de Fluidanimate en un supuesto con ficheros con una tasa de partículas superior la de los ficheros originales. El procedimiento para conseguir esto es sencillo, se ha conservado la dimensión de la matriz tridimensional pero se han modificado los ficheros originales replicando partículas dentro de las celdas, quedando celdas vacías o celdas con el máximo de partículas posibles, 16. En la Tabla 20 se puede comparar la modificación introducida en los casos de estudio utilizados en estas pruebas.

	Nº original de partículas	% de ocupación original	Nº modificado de partículas	% de ocupación modificado
Simsmall	35.402	5,99%	75.712	12,82%
Simmedium	102.850	13,94%	327.680	44,44%
Simlarge	305.809	14,11%	981.824	45,31%
Native	501.642	13,63%	1.645.600	44,73%

Tabla 20. Nº de partículas de los ficheros originales y modificados

Arquitectura unisocket con memoria uniforme

En la Gráfico 15 se representan los tiempos obtenidos para los ficheros *simsmall* y *simmedium* modificados para la arquitectura UMA. Pese a mantener la misma dimensión de la matriz, la aceleración obtenida es bastante superior en esta prueba, si comparamos la aceleración obtenida para el fichero *simsmall* original y el modificado hay un incremento de la aceleración de un 0,63, siendo de 2,08 y 2,71 respectivamente. Esta mejora en el rendimiento se produce incluso con un 12,82% del porcentaje de ocupación, la cual es una cifra relativamente baja. En este caso, el rendimiento de la aplicación con un único hilo ya si es menor que la versión secuencial, por lo que ArBB si consigue conseguir una pequeña optimización.

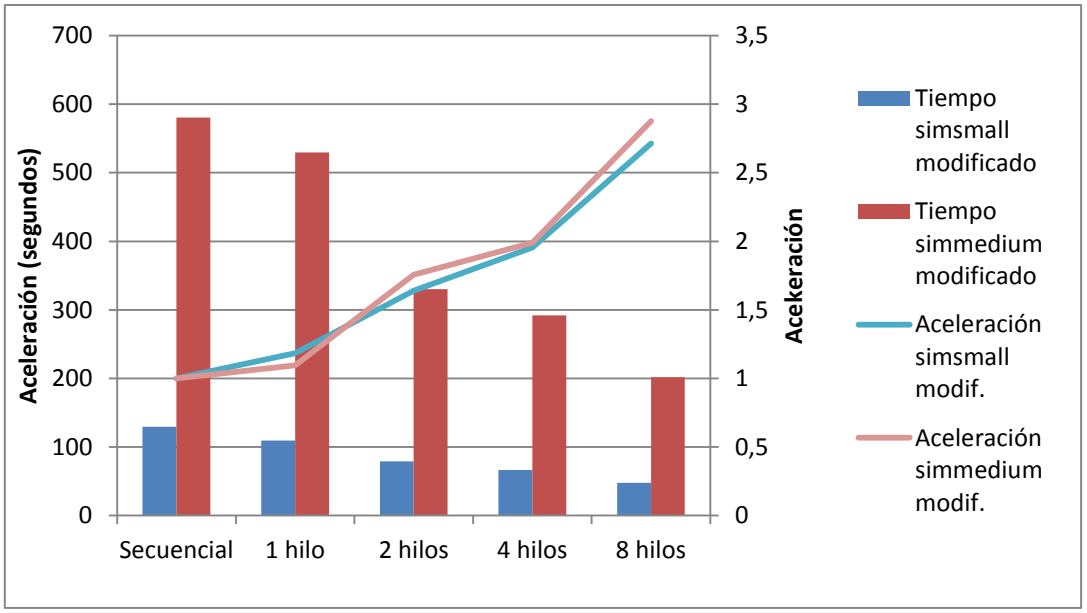


Gráfico 15. Fluidanimate: Tiempos Intel Core i7 para simsmall y simmedium modificados

Centrándose en los ficheros de gran tamaño y con una mayor densidad de partículas también se produce un incremento de la aceleración. En este caso, el incremento es algo menor al ejemplo anterior, pero las aceleraciones conseguidas son de 3,06 y 3,21 para los nuevos ficheros *simlarge* y *native*, introduciendo una mejora de en torno a un 0,3 respecto a las entradas originales. Esta información está detallada en el Gráfico 16.

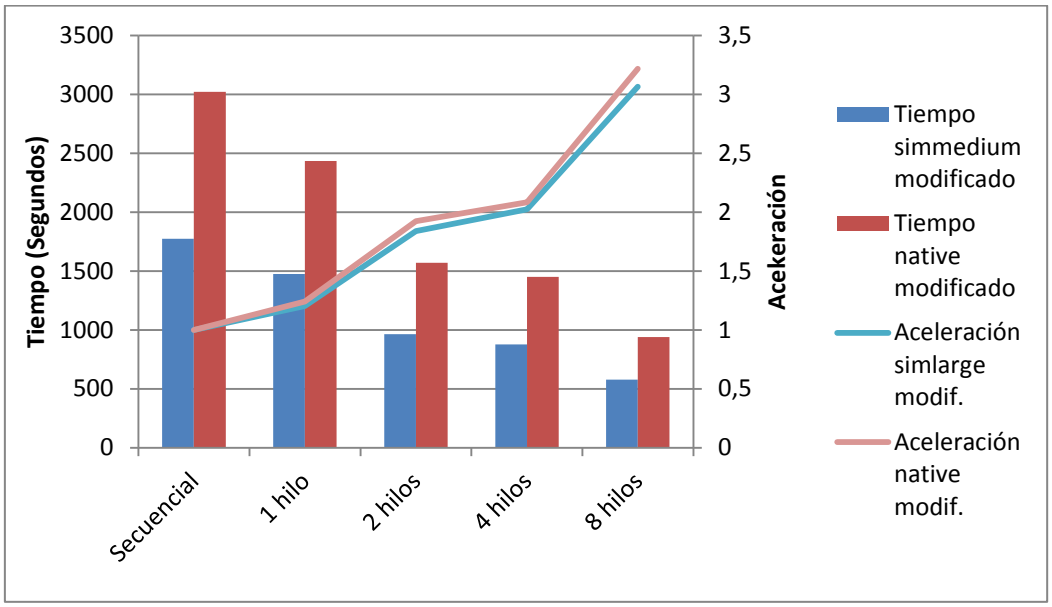


Gráfico 16. Fluidanimate: Tiempos Intel Core i7 para simlarge y native modificados

Arquitectura multi-socket con memoria no uniforme

Los resultados obtenidos en la arquitectura multi-socket son bien distintos a la arquitectura anterior. El computador con un procesador uni-socket solo cuenta con 8 hilos de ejecución lo que no permite un amplio margen de mejora. En este caso, contando con 48 hilos las diferencias entre los ejemplos con ficheros originales y las nuevas entradas modificadas son abismales. En el Gráfico 17 se observan aceleraciones de 10,65 y de 15,25 para los ficheros *simsmall* modificado y *simmedium* modificado respectivamente. La diferencia, respecto a las entradas originales, en cuanto a la aceleración es bastante extensa, situándose cerca del doble. La aceleración para el fichero original *simsmall* era de 5, 34 y para el fichero *simmedium* era de 8,41 (Gráfico 12). Otro dato interesante es la similitud de ambas gráficas en cuanto a la diferencia entre ambos ficheros, en los dos gráficos la diferencia entre ambas no es muy notable hasta que con 8 hilos la pendiente la aceleración para el fichero *simmedium* modificado se hace más pronunciada debido al menor límite computacional del fichero *simsmall*. El último dato reseñable es la utilización de todos los recursos disponibles que en la prueba realiza con los ficheros originales no se lograba conseguir.

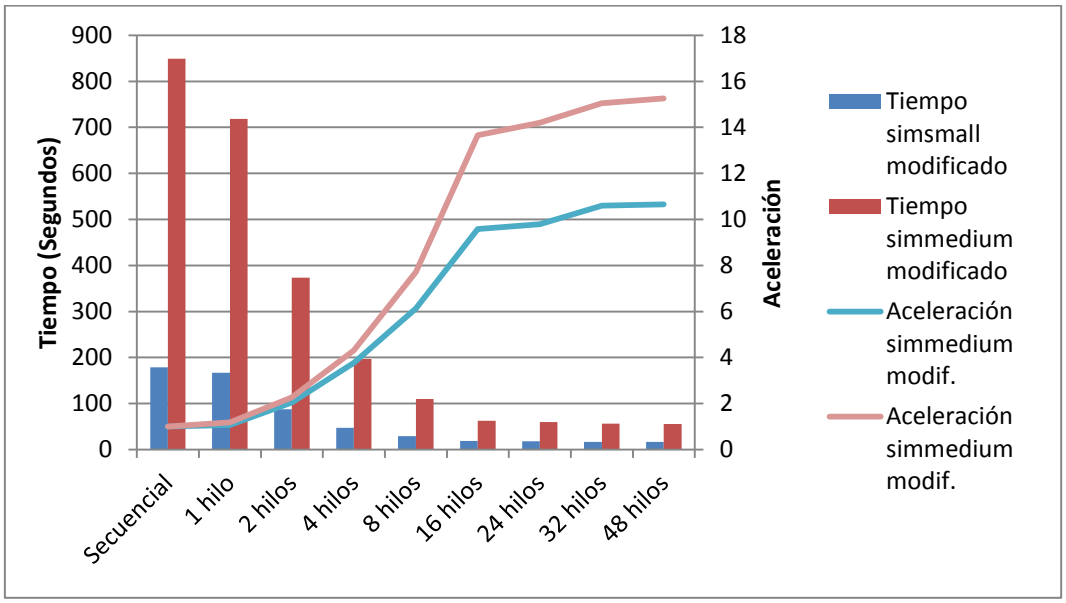


Gráfico 17. Fluidanimate: Tiempos Intel Xeon E7 para simsmall y simmedium modificados

En esta última prueba, queda por medir los tiempos obtenidos para ficheros de gran dimensión con un alto nivel de partículas en una arquitectura NUMA. Los resultados son similares a los del caso anterior. En ambos casos de estudio se introducen mejoras muy notables en cuanto al rendimiento, el Gráfico 18 muestra una aceleración máxima de 17,96 y de 18,45 para los ficheros *simlarge* y *native* modificados, superando los 8,43 y 8,49, de la prueba con los ficheros originales por más del doble. Tanto en esta medición como en el caso anterior, a partir de los 16 hilos de ejecución el rendimiento se

incrementa más lentamente, por lo que es hasta los 16 hilos cuando se consiguen las mejores aceleraciones por core (eficiencia).

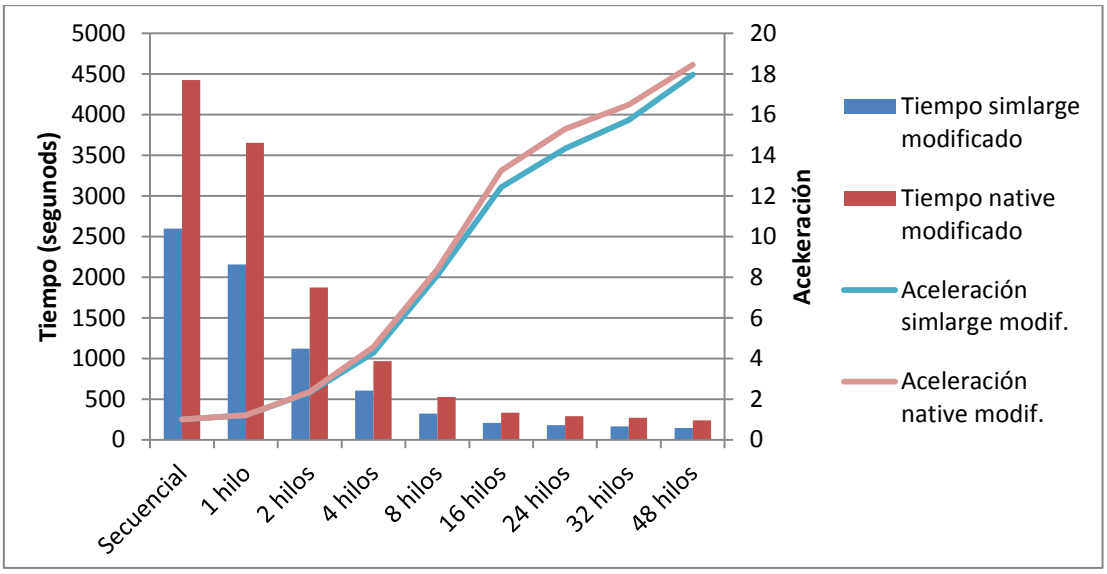


Gráfico 18. Fluidanimate: Tiempos Intel Xeon E7 para simlarge y native modificados

5.4.2.4. Comparación entre arquitecturas

Tras mostrar los tiempos medidos con los diferentes casos de estudios es interesante comprobar la eficiencia de la aplicación acelerada en cada arquitectura. Para comprobar este dato se hace uso de la ecuación mostrada en el punto 2.2 para el cálculo de la aceleración por core.

En el Gráfico 19 se muestran las aceleraciones por core para cada prueba realizada. Para la arquitectura UMA el número de hilos son 8 y para la arquitectura NUMA el número de hilos es 48. Del gráfico pueden extraerse dos clases de resultados: en los que Intel Core i7 obtiene una gran ventaja y en los que ambas arquitecturas son similares con una pequeña diferencia en contra para el Intel Xeon E7.

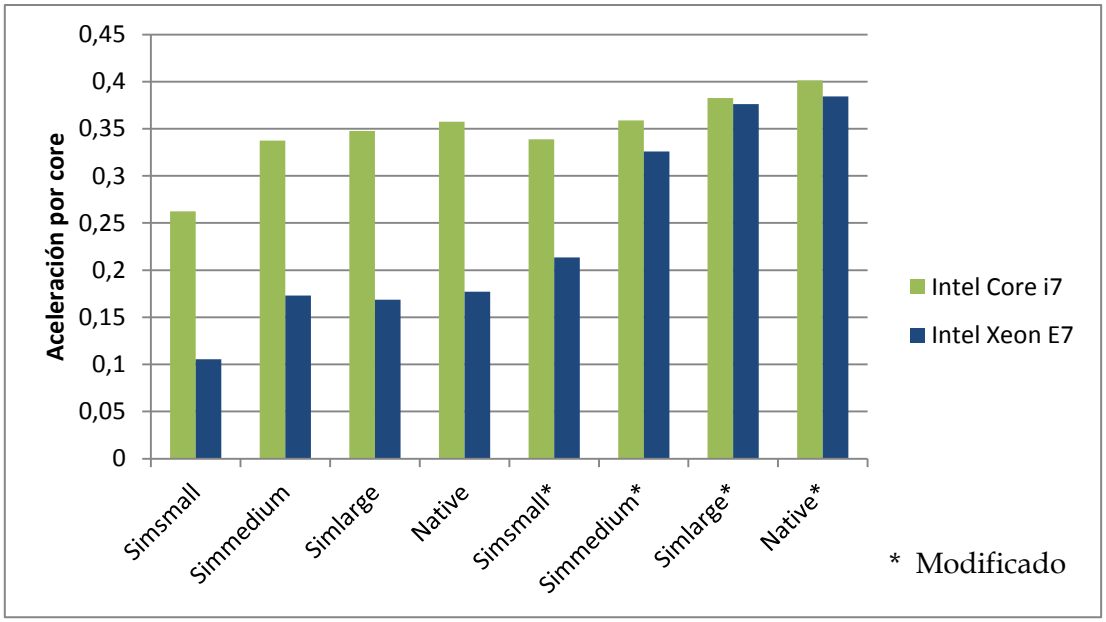


Gráfico 19. Fluidanimate: Evaluación de la eficiencia

En esta aplicación la influencia de los fallos de caché no es tan elevada como en Blackscholes, los resultados obtenidos en los que el Intel Core i7 es más eficiente se deben únicamente al número de cores de cada arquitectura. La carga de trabajo para los primeros ficheros utilizados no es muy elevada, y por tanto la granularidad del trabajo a partir de los 8-16 hilos de ejecución es demasiado fina impidiendo un crecimiento de la aceleración proporcional, esta información puede corroborarse en las gráficas 12,14 y 17.

La otra clase de resultados en los que el Intel Core i7 saca una mínima ventaja a la otra arquitectura utilizada pertenecen a las pruebas realizadas con una gran necesidad de cómputo, estos ficheros si permiten una escalabilidad mayor de su rendimiento y por tanto consiguen maximizar la aceleración incluso con 48 hilos de ejecución.

5.4.2.5. Comparación con el resto de modelos de programación paralela

En esta última prueba, se han agrupados los resultados obtenidos en los puntos anteriores y se han comparado con las versiones de Fluidanimate implementadas con Pthreads e Intel TBB. El Gráfico 20 muestra toda la información recopilada. Muchos son los datos interesantes que se pueden extraer de dicha gráfica.

Primeramente van a pasar a destacarse las comparaciones en las que la versión con Intel TBB y/o Pthreads obtienen ventaja frente a la versión implementada. El primero de ellos es el caso de estudio *simsmall* en la arquitectura UMA, donde la aceleración es de 2,4 y de 2,01 para Intel TBB y la versión implementada respectivamente. En este caso, Intel TBB obtiene mejores resultados debido a que el fichero *simsmall* cuenta con

la tasa de ocupación más baja de todos los casos de estudios analizados y es la única prueba en la que la versión con ArBB introducía pérdidas de rendimiento con un único hilo. Las siguientes comparaciones en la que Intel TBB y Pthreads ganan a la aplicación implementada pueden agruparse en un mismo grupo, se tratan de las pruebas realizadas con los ficheros originales en la arquitectura NUMA. Pese a que la versión implementada de Fluidanimate permitía obtener mejores resultados que la versión implementada de Blackscholes debido a la fusión de Intel ArBB e Intel TBB y por tanto a la reducción de la tasa de fallos de caché, la utilización de ArBB supone un incremento de la tasa de fallos de caché y si se compara con una versión íntegra con Intel TBB o Pthreads junto con unas características desfavorables para ArBB (porcentaje de ocupación bajo) la versiones incluidas originalmente en el benchmark PARSEC son claras vencedoras en esta comparación.

El resto de comparaciones se decantan por la versión implementada con ArBB y TBB conjuntamente. Las diferencias entre las aceleraciones obtenidas en las pruebas con los ficheros originales en la arquitectura UMA son pequeñas pero favorables a la versión con ArBB. En los ficheros modificados, la diferencia de la aceleración es más notable situándose entre el 0,6 y el 0,8 aproximadamente. Por último, la diferencia de aceleraciones en la arquitectura NUMA para los ficheros con una alta densidad de ocupación está próxima a uno lo que otorga una mayor ventaja a la versión implementada. En el gráfico mostrado puede observarse la diferencia entre los ficheros originales y los modificados donde, principalmente en la arquitectura NUMA, las diferencias son altas.

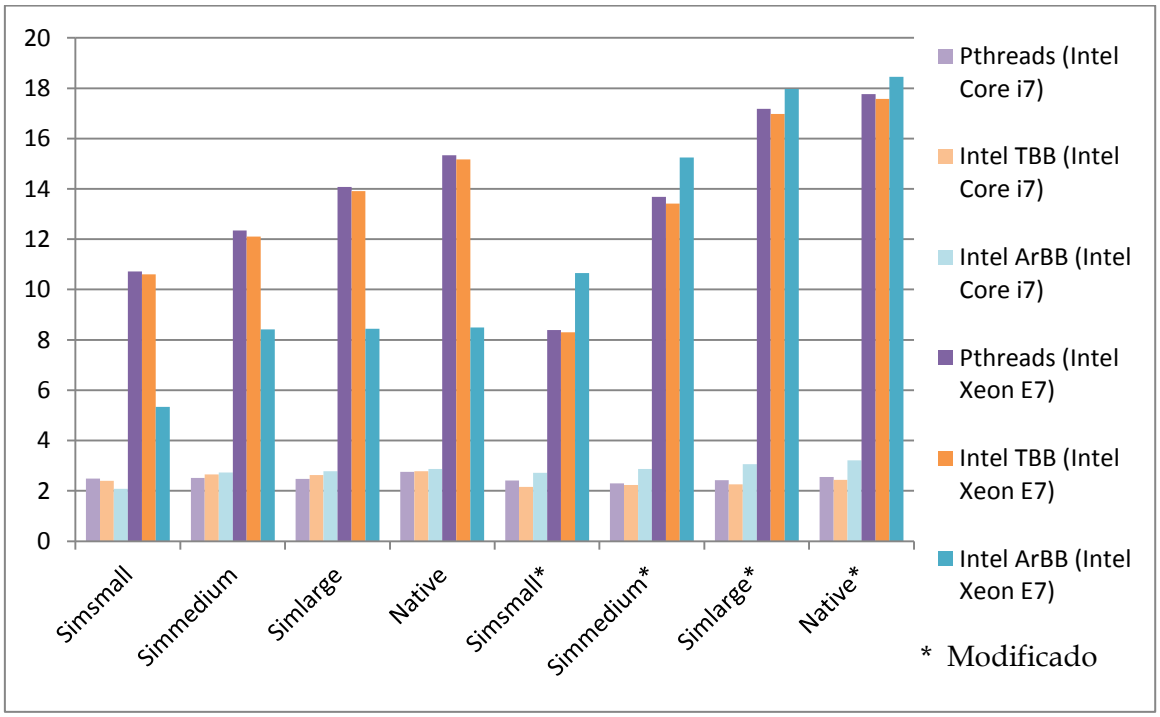


Gráfico 20. Fluidanimate: Resumen comparativo

6. CONCLUSIONES

6.1.Verificación de requisitos

A continuación se muestra un resumen del resultado de la verificación de los requisitos planteados en el punto 3.1. En la Tabla 21 se describe esta información.

Requisito	Resumen	Resultado	Observaciones
RNF-01	Desarrollo y evaluación en Linux	✔	-
RNF-02	Seleccionar las aplicaciones del benchmark más afines a ArBB	✔	-
RNF-03	Las aplicaciones del PARSEC compatibles con Linux y gcc-4.4.	✔	-
RNF-04	Permitir utilizar varios cores en las aplicaciones implementadas	✔	-
RNF-05	Compromiso máximo de 70%-30% entre Intel TBB e Intel ArBB	✔	Solo tres de las siete funciones de Fluidanimate han sido paralelizadas haciendo uso de Intel TBB
RNF-06	Evaluación del rendimiento en distintas máquinas	✔	-
RNF-07	Aceleración, respecto la versión secuencial, de 2 o superior.	✔	-
RNF-08	Comparar las aplicaciones con el resto de modelos	✔	La versión de OpenMP de Fluidanimate no ha sido evaluada debido a que aún no ha sido incorporada al PARSEC de forma oficial
RF-09	Mismos argumentos para todas las versiones	✔	-
RF-10	Mismos resultados para todas las versiones	✔	Ambas aplicaciones tienen un ligero desvío de los resultados que puede clasificarse como irrelevante

Tabla 21. Resumen objetivos cumplidos

6.2.Líneas futuras

Cómo próximos trabajos futuros, propongo continuar el estudio de Array Building Blocks en otro tipo de aplicaciones que requieran un alto coste computacional, esto permitirá obtener una visión global de las características y limitaciones de este novedoso modelo de programación paralela.

En lo referente a Blackscholes y Fluidanimate, un estudio interesante sería la evaluación del código implementado en otros sistemas que sí posean las instrucciones vectoriales AVX incluidas en la familia de procesadores Sandy Bridge de Intel. Con estas instrucciones podría conseguirse una mayor aceleración.

Blackscholes y Fluidanimate han sido paralelizadas en varios modelos de programación paralela como OpenMP, Intel TBB o Intel ArBB, no obstante, una buena línea de trabajo futura sería la paralelización de estas aplicaciones haciendo uso de CUDA (*Compute Unified Device Architecture*)[21] y C++ AMP (*Accelerated Massive Parallelism*)[22] para poder realizar una evaluación más completa de todos los modelos de programación paralela del momento en arquitecturas de memoria compartida.

La arquitectura CUDA diseñada por NVidia es un entorno de trabajo que permite la ejecución en paralelo mediante dispositivos GPU para C/C++ inicialmente. Estos dispositivos permiten el lanzamiento de centenares de hilos de ejecución simultáneos, lo que da lugar a una gran división del trabajo en muchos hilos de ejecución. Similar a CUDA, existe la biblioteca C++ AMP de Microsoft, que consigue reducciones en el tiempo de ejecución para códigos escritos en C++. Este modelo permite la paralelización de datos en dispositivos GPUs y facilita al desarrollador programar sobre este tipo de dispositivos de forma sencilla.

6.3.Conclusión personal

Como conclusión quiero manifestar mi opinión acerca del trabajo realizado. Desde mi punto de vista, Intel Array Building Blocks ofrece un novedoso modelo de programación paralela haciendo uso de las instrucciones vectoriales y de los distintos cores de la máquina en cuestión. Además de esto, la paralelización de una aplicación había venido siendo una tarea complicada para los programadores, por lo que Intel ha invertido parte de sus esfuerzos en simplificar la programación en este tipo de tareas. Gracias a estas dos cualidades, ArBB ofrece un gran rendimiento y una simplicidad mayúscula si lo comparamos con otros modelos como Intel TBB u OpenMP. Creo que el mejor ejemplo de esto es el trabajo realizado sobre Blackscholes donde tanto el rendimiento como la tarea de diseño e implementación han sido muy favorables.

Sin embargo, considero que ArBB tiene grandes limitaciones en cuanto al alcance de su utilidad. Pese a tener un gran rendimiento en operaciones vectoriales como la

multiplicación de matrices, cuando la aplicación a paralelizar es de ámbito más general y su algoritmo difiere mínimamente del esperado, la optimización resulta casi imposible. Ejemplo de esto son las funciones *ComputeDensities()* y *ComputeForces()* de Fluidanimate que, tras muchos intentos en vano, determinamos que ArBB era incapaz de optimizar estas funciones y me vi obligado a utilizar otra alternativa como es Intel TBB. Puede que este último modelo de programación paralela no sea tan eficiente como ArBB pero su alcance es mucho mayor.

Otro de los problemas que, a mi juicio, presenta ArBB es la alta abstracción con la que cuenta el programador. Esta característica puede ser considerada como un arma de doble filo, puesto que la programación puede ser más sencilla pero para conseguir esto es necesario utilizar una nueva sintaxis e imponer grandes restricciones, lo que priva al desarrollador de poder adaptar ArBB a las necesidades específicas de la aplicación. Un ejemplo de esto la restricción de operar sobre la totalidad de la matriz.

Finalmente, pienso que el margen de mejora de ArBB es aún bastante amplio. El lanzamiento de este modelo es muy reciente, se encuentra en fase beta y aún queda trabajo por hacer. Si funciones como *position()* o estructuras como los *nested* no implicasen tanta sobrecarga en la aplicación, Fluidanimate podría haber sido paralelizada íntegramente con ArBB.

Tras recopilar todas estas ventajas y desventajas de ArBB concluyo afirmando que estos nuevos modelos de programación paralela tienen aún una gran progresión, ya que intentan explotar todos los recursos del computador no limitándose a la división del trabajo en varios procesadores.

En cuanto a mis objetivos, este Trabajo Fin de Grado me ha sido útil para adquirir cierto dominio sobre el lenguaje C++ que, hasta el momento de iniciación del trabajo, me era desconocido. La realización de este proyecto ha supuesto no solo el acercamiento a Intel Array Building Blocks, sino al conjunto de modelos paralelos, ya que para las tareas de comparación de resultados es necesario poseer un conocimiento de todas las tecnologías utilizadas. El hardware ha sido otro de los principales protagonistas en este trabajo puesto que, tanto la motivación del proyecto como la evaluación de los resultados en las distintas arquitecturas, exigían de una noción mínima previa que durante la carrera había obviado en muchas ocasiones. La concepción de software paralelo no tiene sentido sin una base hardware de calidad y creo que esa es una de las máximas que me ha aportado este trabajo.

7. PLANIFICACIÓN DEL TRABAJO

Para la correcta ejecución del Trabajo Fin de Grado y la estimación de plazos, recursos y presupuesto, se hace un estudio inicialmente y se estructuran las tareas. Para una gestión eficiente, las tareas más heterogéneas han sido divididas en subtareas. En la Tabla 22 se muestran las tareas planificadas inicialmente y en la Ilustración 18 se representa esta planificación mediante un diagrama de Gantt, para facilitar la lectura se han dividido las tareas en colores siendo verde para las tareas generales del trabajo, azul aprendizaje de ArBB y TBB, morado para Blackscholes y naranja para Fluidanimate. La fecha de finalización del TFG para el día 08/06/2012.

Tarea	Subtareas	Descripción
Instalación y configuración del entorno de trabajo		Acondicionamiento del computador de desarrollo (instalación del SO, configuración de ArBB, IDE de desarrollo, etc.).
Aprendizaje de Intel ArBB		Estudio de las características de ArBB y de su sintaxis.
Benchmark PARSEC	• Instalación y configuración del benchmark. • Evaluación de las aplicaciones. • Informe de clasificación de las aplicaciones y elección de aplicaciones.	Instalación del benchmark incluyendo un análisis de las aplicaciones y un informe seleccionando las aplicaciones candidatas.
Fluidanimate	• Análisis de la aplicación. • Diseño de la aplicación. • Implementación. • Medición y evaluación de los resultados.	Paralelización de la aplicación Fluidanimate incluyendo todas las fases de desarrollo.
Toma de tiempos definitiva		Medición del rendimiento de Fluidanimate con distintas arquitecturas y modelos de programación paralela.
Redacción de la memoria del TFG		Documentación del trabajo realizado

Tabla 22. Tareas en la planificación inicial

Id.	Nombre de tarea	Comienzo	Fin	Duración
1	Instalación y configuración del entorno de trabajo	08/12/2011	19/12/2011	10d
2	Aprendizaje de Intel ArBB	12/12/2011	26/01/2012	40d
3	**Benchmark PARSEC**	**26/01/2012**	**23/02/2012**	**25d**
4	Instalación y configuración del benchmark	26/01/2012	31/01/2012	5d
5	Evaluación de las aplicaciones	01/02/2012	17/02/2012	15d
6	Informe de clasificación y elección de las aplicaciones	18/02/2012	23/02/2012	5d
7	**Fluidanimate**	**23/02/2012**	**05/04/2012**	**37d**
8	Análisis de la aplicación Fluidanimate	23/02/2012	05/03/2012	10d
9	Diseño la aplicación Fluidanimate con ArBB	06/03/2012	16/03/2012	10d
10	Implementación de Fluidanimate con ArBB	17/03/2012	03/04/2012	15d
11	Medición y evaluación de los resultados de Fluidanimate con ArBB	04/04/2012	05/04/2012	2d
12	Toma de tiempos final	05/04/2012	12/04/2012	7d
13	Redacción de la memoria del TFG	12/04/2012	08/06/2012	50d

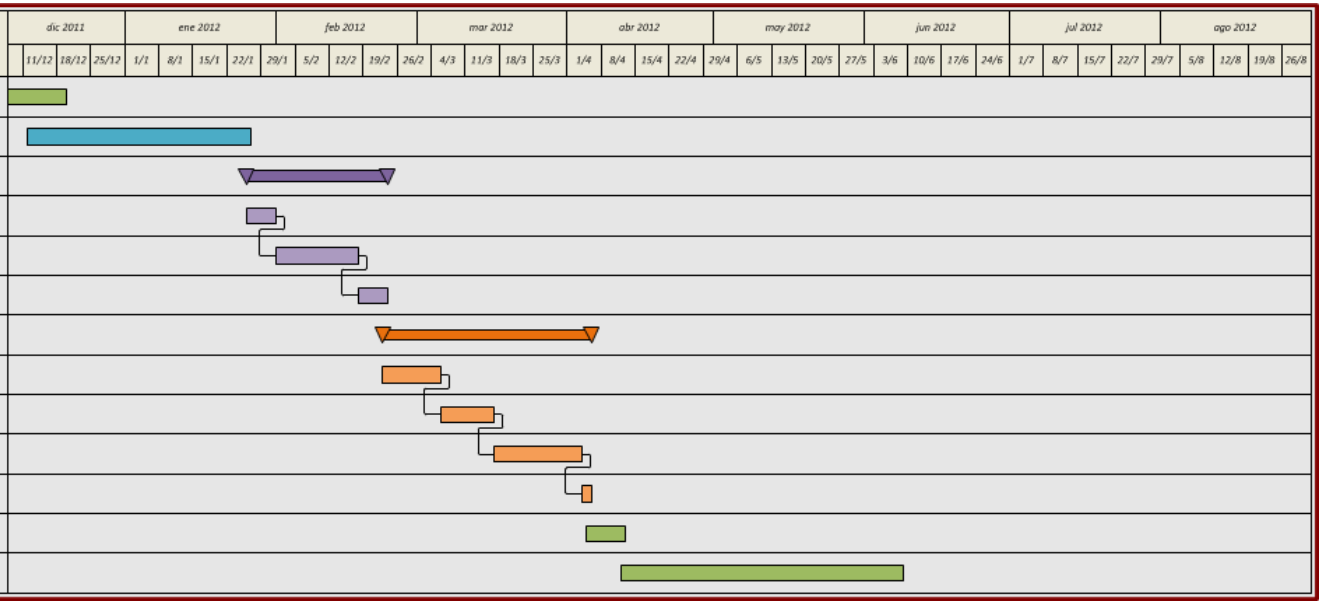


Ilustración 18. Diagrama de Gantt inicial

Debido a la gran complejidad en la paralelización de Fluidanimate y frente a la imposibilidad de cumplir los plazos, se aplazó la fecha de finalización del TFG al 31/08/2012. En esta planificación final se añadió una nueva aplicación con sus respectivas tareas como se muestra en la Tabla 23. En el diagrama de Gantt de la Ilustración 19 se muestra la planificación final con pequeñas fluctuaciones hasta el mes de marzo.

Tarea	Subtareas	Descripción
Instalación y configuración del entorno de trabajo		Acondicionamiento del computador de desarrollo (instalación del SO, configuración de ArBB, IDE de desarrollo, etc.).
Aprendizaje de Intel ArBB		Estudio de las características de ArBB y de su sintaxis.
Benchmark PARSEC	• Instalación y configuración del benchmark. • Evaluación de las aplicaciones. • Informe de clasificación y elección de aplicaciones.	Instalación del benchmark incluyendo un análisis de las aplicaciones y un informe seleccionando las aplicaciones candidatas.
Fluidanimate (versión 1.0)	• Análisis de la aplicación. • Diseño de la aplicación. • Implementación. • Medición y evaluación de los resultados.	Paralelización de la aplicación Fluidanimate incluyendo todas las fases de desarrollo.
Blackscholes	• Análisis de la aplicación. • Diseño de la aplicación. • Implementación. • Medición y evaluación de los resultados.	Paralelización de la aplicación Blackscholes incluyendo todas las fases de desarrollo.
Fluidanimate (versión 2.0)	• Diseño de la aplicación. • Implementación. • Medición y evaluación de los resultados.	Nuevo diseño con su posterior implementación para Fluidanimate, incluyendo la evaluación de resultados.
Fluidanimate (versión 3.0)	• Diseño I de la aplicación. • Aprendizaje de Intel TBB. • Diseño II de la aplicación. • Implementación. • Medición y evaluación de los resultados.	Última paralelización de Fluidanimate con ArBB, en el que se incluye un nuevo modelo de programación paralela (Intel TBB).
Toma de tiempos definitiva		Evaluación de Fluidanimate con distintas arquitecturas y modelos de programación paralela.
Redacción de la memoria del TFG		Documentación del trabajo realizado

Tabla 23. Tareas en la planificación final

Id.	Nombre de tarea	Comienzo	Fin	Duración
1	Instalación y configuración del entorno de trabajo	01/12/2011	12/12/2011	10d
2	Aprendizaje de Intel ArBB	12/12/2011	23/01/2012	37d
3	**Benchmark PARSEC**	**24/01/2012**	**18/02/2012**	**23d**
4	Instalación y configuración del benchmark	24/01/2012	26/01/2012	3d
5	Evaluación de las aplicaciones	27/01/2012	11/02/2012	14d
6	Informe de clasificación y elección de las aplicaciones	13/02/2012	18/02/2012	6d
7	**Fluidanimate (versión 1.0)**	**18/02/2012**	**31/03/2012**	**37d**
8	Análisis de la aplicación Fluidanimate	18/02/2012	28/02/2012	9d
9	Diseño la aplicación Fluidanimate con ArBB (versión 1.0)	28/02/2012	12/03/2012	12d
10	Implementación de la versión 1.0 de Fluidanimate con ArBB	13/03/2012	27/03/2012	13d
11	Medición y evaluación de los resultados de la versión 1.0 de Fluidanimate	29/03/2012	31/03/2012	3d
12	**Blackscholes**	**31/03/2012**	**28/04/2012**	**25d**
13	Análisis de la aplicación Blackscholes	31/03/2012	12/04/2012	11d
14	Diseño de la aplicación Blackscholes con ArBB (versión 1.0)	13/04/2012	20/04/2012	7d
15	Implementación de la versión 1.0 de Blackscholes con ArBB	21/04/2012	26/04/2012	5d
16	Medición y evaluación de los resultados de la versión 1.0 de Blackscholes	27/04/2012	28/04/2012	2d
17	**Fluidanimate (versión 2.0)**	**28/04/2012**	**21/05/2012**	**20d**
18	Diseño de la aplicación Fluidanimate con ArBB (versión 2.0)	28/04/2012	08/05/2012	9d
19	Implementación de la versión 2.0 para Fluidanimate	08/05/2012	18/05/2012	10d
20	Medición y evaluación de los resultados de la versión 2.0 de Fluidanimate	19/05/2012	21/05/2012	2d
21	**Fluidanimate (versión 3.0)**	**21/05/2012**	**05/07/2012**	**40d**
22	Diseño I de la aplicación Fluidanimate con ArBB/TBB(versión 3.0)	21/05/2012	24/05/2012	4d
23	Aprendizaje de Intel TBB	24/05/2012	02/06/2012	9d
24	Diseño II de la aplicación Fluidanimate con ArBB/TBB(versión 3.0)	04/06/2012	14/06/2012	10d
25	Implementación de la versión 3.0 para Fluidanimate	15/06/2012	02/07/2012	15d
26	Medición y evaluación de los resultados de la versión 3.0 de Fluidanimate	03/07/2012	05/07/2012	3d
27	Toma de tiempos final: versión 1.0 de Blackscholes y versión 3.0 de Fluidanimate	06/07/2012	17/07/2012	10d
28	Redacción de la memoria del TFG	02/07/2012	30/08/2012	52d

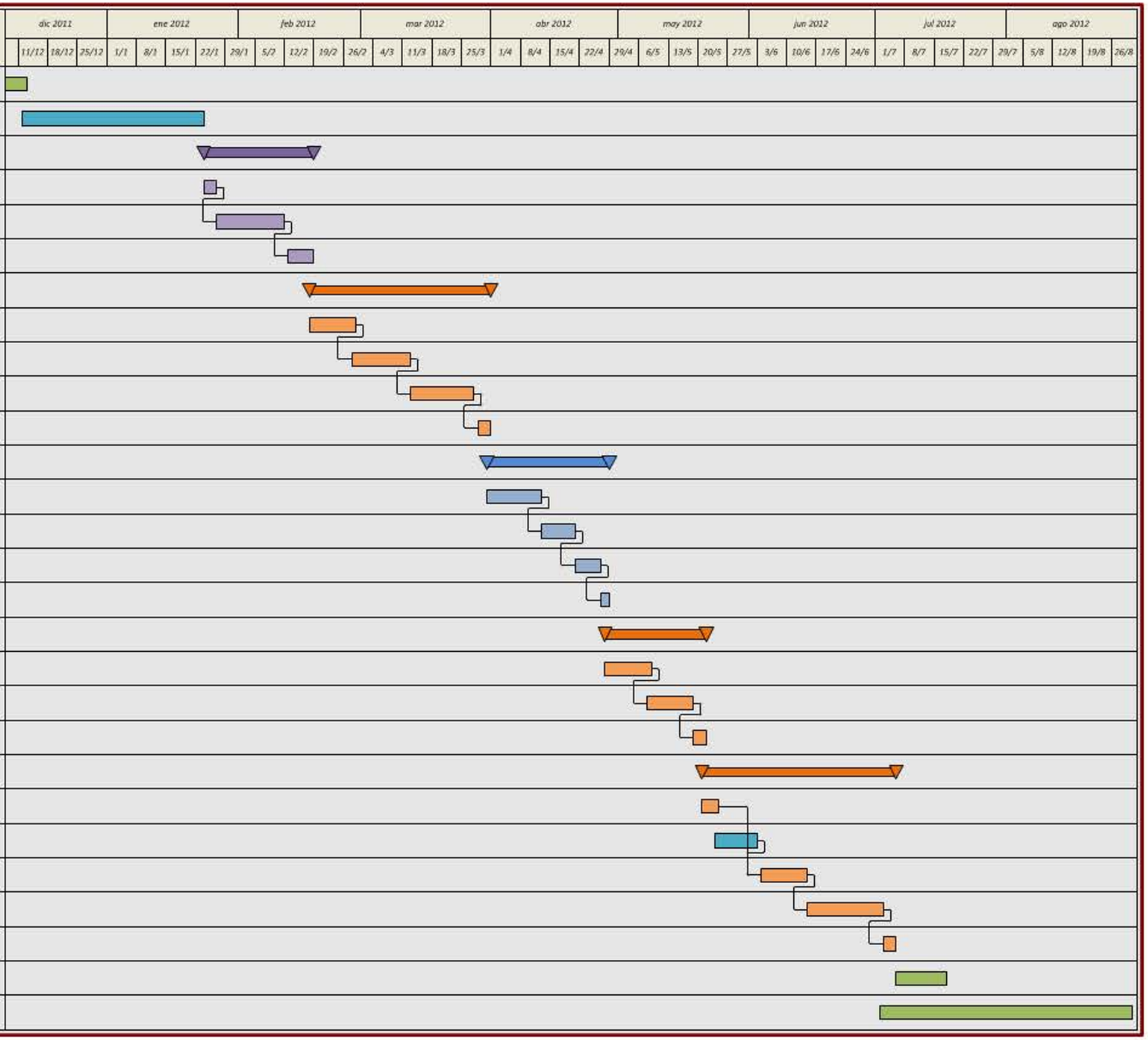


Ilustración 19. Diagrama de Gantt real

8. PRESUPUESTO

A continuación se expone el presupuesto calculado para la realización de este Trabajo Fin de Grado. Para el cálculo se ha como referencia la planificación del punto 7 con un total de 254 días a una media de 20-25 horas semanales, los costes de personal pueden observarse en la Tabla 24. Dado que el proyecto ha sido desarrollado con fines académicos y de investigación no se han aplicado porcentajes extra por riesgos y beneficios en los costes del personal. Para los recursos hardware y software se ha calculado la amortización en función de los meses en los que han sido utilizados, estos costes pueden observarse en la Tabla 25. La suma total del presupuesto está descrita en la Tabla 26, siendo la cuantía final de veinte mil cuatrocientos sesenta y seis euros.

Puesto	Horas totales	Coste/hora	Coste total
Ingeniero de desarrollo	800	25€	20.000€
Coste: recursos humanos			**20.000€**

Tabla 24. Costes recursos humanos

Elemento	Coste	Tiempo de vida	Coste mensual	Meses utilizado	Coste amortizado
Hardware					
Asus N61J	799 €	36	22 €	9	200 €
Compute-7-2	8.000,00 €	36	222 €	1	222 €
Coste hardware					**422 €**
Software					
Licencia Windows 7	164,00 €	50	3 €	6	20 €
Licencia Microsoft Office 2010	199,00 €	50	4 €	6	24 €
Software libre	0,00 €	50	0 €	9	0 €
Coste software					**32 €**
Coste: hardware y software					**466 €**

Tabla 25. Costes hardware y software

Elemento	Coste total
Recursos humanos	**20.0000 €**
Hardware y software	**466 €**
Coste total	**20.466€**

Tabla 26. Presupuesto

9. REFERENCIAS

[1] Herb Sutter .The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 2005.

[2]Christian Bienia, Sanjeev Kumar, Jaswidner Pal Singh and Kai Li. Technical. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Princeton University, 2008.

[3] P. Dubey. A Platform 2015 Workload Model Recognition, Mining and Synthesis Moves Computers to the Era of Tera. 2005

[4] PARSEC Wiki (2012, July). [Online]. Available: http://wiki.cs.princeton.edu/index.php/PARSEC

[5] P. Lamothe Fernández and M. Pérez Somalo. Opciones financieras y productos estructurados, segunda edición ed. McGraw-Hill. 2003.

[6] G. Batchelor. An Introduction to Fluid Dynamics, Cambridge Mathematical Library edition, 2000.

[7] Fluidanimate 35K Particles.wmv. (2012,August). [Online]. Available: http://www.youtube.com/watch?v=nPOcffzNAy8

[8] S. M. Faizur Rahman, Q. Yi and A. Qasem. Understanding Stencil Code Performance On Multicore Architectures. CF '11 Proceedings of the 8th ACM International Conference on Computing Frontiers, 2011.

[9] Liu Peng, Richard Seymour, Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R. Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing (23rd IPDPS'09), 2009.

[10] D. A. Patterson and J. L. Hennessy. Computer Architecture A Quantitative Approach, Fourth edition, ed. Morgan Kaufmann, 2007.

[11] D. A. Patterson and J. L. Hennessy. Computer Organization and Design, Fourth edition, ed. Morgan Kaufmann, 2009.

[12] Livermore National Laboratory Lawrence Blaise Barney. POSIX Threads Programming.

[13] B. Chapman, G. Jost, and R. v. d. Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, 2007.

[14] James Reinders. Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism. First edition, ed. O'Reilly, 2007.

[15] Intel® TBB. (2012, July) [Online]. Available: http://threadingbuildingblocks.org

[16] C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z.-G. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang Intel® Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language". CGO, 2011.

[17] Intel ® Vtune ™ Amplifier XE. (2012, June). Available: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

[18] GNU gprof. (2012, May) [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

[19] M. Klemm, M. McCool. SC10 Tutorial: Using Intel® Array Building Blocks for Efficient Development of Multicore Applications. International Conference for High Performance Computing, Networking, Storage, and Analysis, 2010.

[20] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys, 1991

[21] Jason Sanders, Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, first edition ed. Addison-Wesley, 2010

[22] Microsoft Corporation. C++ AMP: Language and Programming Model. Version 0.99 May 2012.